O'REILLY®

TURING 图灵程序设计丛书

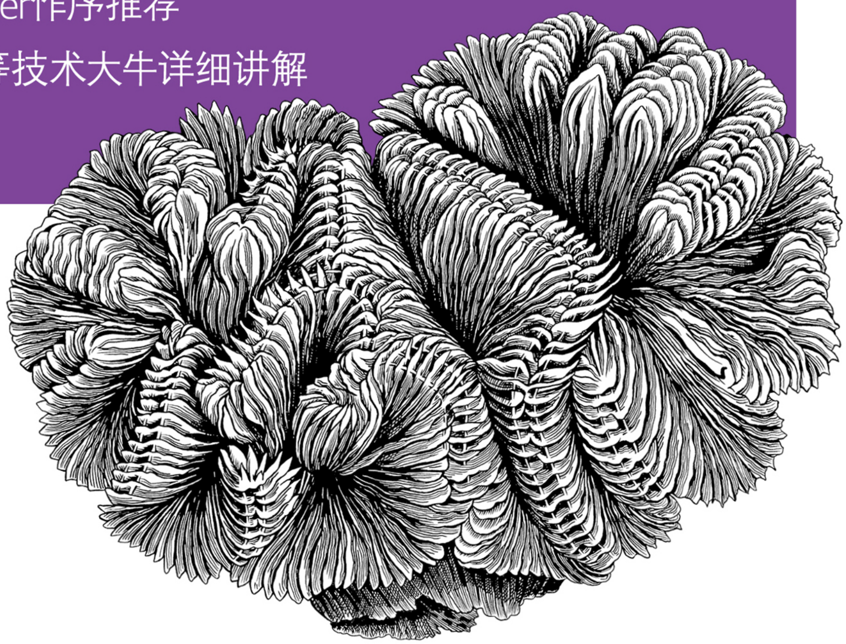
演进式架构

Building Evolutionary Architectures

敏捷之父Martin Fowler作序推荐

ThoughtWorks CTO等技术大牛详细讲解

先进架构思想



[美] 尼尔·福特　丽贝卡·帕森斯 著

[澳] 帕特里克·柯

周训杰 译

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

周训杰

ThoughtWorks高级咨询师，拥有十余年企业级应用和互联网应用的开发和架构经验，在ThoughtWorks带领多个团队完成不同规模的海外项目交付，目前负责大型企业级项目的服务化平台搭建及技术团队管理。

数字版权声明

TURING 图灵程序设计丛书

演进式架构

Building Evolutionary Architectures

[美] 尼尔·福特 [美] 丽贝卡·帕森斯 [澳] 帕特里克·柯 著
周训杰 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

演进式架构 / (美) 尼尔·福特 (Neal Ford) ,
(美) 丽贝卡·帕森斯 (Rebecca Parsons) , (澳) 帕特
里克·柯 (Patrick Kua) 著 ; 周训杰译. -- 北京 : 人
民邮电出版社, 2019.8
（图灵程序设计丛书）
ISBN 978-7-115-51617-6

Ⅰ. ①演… Ⅱ. ①尼… ②丽… ③帕… ④周… Ⅲ.
①程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2019)第137818号

内 容 提 要

在软件开发流程中，为了尽可能快地响应各种变化，理应把结构渐进改变作为设计的首要原则。本书详尽阐述了演进式架构的必要性、构建方法以及需要注意的问题。各章结合案例分别讨论了软件架构、适应度函数、开展增量变更、架构耦合、演进式数据、构建可演进的架构、演进式架构的陷阱和反模式，以及实践演进式架构。

本书适合所有架构师及开发人员阅读。

◆ 著　　[美] 尼尔·福特 [美] 丽贝卡·帕森斯
　　　　[澳] 帕特里克·柯
译　　周训杰
责任编辑　朱　巍
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：9.75
字数：230千字　　　　2019年 8 月第 1 版
印数：1－3 500册　　　　2019年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-8084号

定价：59.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务还是面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

序

长久以来，软件行业都奉行这样一个理念：在开始编写第一行代码前就应该完成架构开发。受到建筑行业的影响，人们认为成功的软件架构在开发过程中不需要修改，而且重新架构往往会导致高成本的报废和返工。

随着敏捷软件开发方法的兴起，这样的架构愿景受到了很大的挑战。预先规划的架构要求在编码前就确定需求，因而催生了分阶段（即瀑布式）的开发方法——在完成需求分析后才开始设计架构，然后再进入构建（编码）阶段。然而，敏捷软件开发方法并不认同"需求是确定的"这样的观点。它发现现代商业中需求不断变化是必然的，并进一步提供了项目规划技术来拥抱受控的变化。

在这个新的敏捷世界里，很多人质疑架构的作用。当然，预先规划的架构无法适应动态的现代业务，但是，还是会有另外一种架构，它以敏捷的方式拥抱变化。如此看来，架构是不断努力的结果，是一个与开发工作紧密结合的过程，这样它才能同时响应不断变化的需求和开发人员的反馈。我们称之为"演进式架构"，正是要强调当无法预测变化时，该架构仍然可以朝着正确的方向发展。

在 ThoughtWorks，我们无时无刻不秉持演进式架构的理念。Rebecca 在 21 世纪初领导了多个重要项目，作为 CTO，她提升了 ThoughtWorks 的技术领导力。Neal 一直密切关注着我们的工作，将我们学到的经验汇总并传播开来。Pat 在开展项目工作的同时也加强了我们的技术领导力。我们始终认为架构是极其重要的，不容忽视。我们虽然也犯过错误，但我们从错误中学习，更好地理解了如何构建出能正确应对各种变化的系统。

构建演进式架构的核心是采取小步变更，然后通过反馈环让团队的每个成员不断地从系统的发展过程中学习。持续交付的兴起使得演进式架构变得切实可行。三位作者通过适应度函数监控架构的状态。他们探索了架构演进的不同方式，并且重视那些通常被别人所忽视的长存数据。在讨论中也理所当然地会经常提及康威定律。

关于构建演进式软件架构，虽然我们还需要了解很多东西，但本书基于当前的理解给出了基本线路图。随着越来越多的人意识到软件系统在 21 世纪人类社会中的核心地位，每个软件领导者都应该知道如何在发展中以最好的姿态应对变化。

Martin Fowler

前言

排版约定

本书使用如下排版约定。

- **黑体字**
  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）
  表示程序以及段落内引用的程序元素，如变量、函数名、数据库、数据类型、环境变量、语句和关键字。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）
  表示应该被替换为由用户提供的值或由上下文确定的值的文本。

该图标表示提示或建议。

O’Reilly Safari

Safari（原来叫 Safari Books Online）是一个会员制的培训和参考咨询平台，面向企业、政府、教育从业者和个人。

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动式教程和精选播放列表，这些出版商包括 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

要了解更多信息，可以访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到书的相关信息，包括勘误表[1]、示例代码以及其他信息。本书的网站地址是：http://oreil.ly/2eY9gT6。

对于本书的评论和技术性问题，请发送电子邮件到 bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

附加信息

可以通过 http://evolutionaryarchitecture.com 访问本书的配套站点。

注 1：本书中文版的勘误请到 http://www.ituring.com.cn/book/2440 查看和提交。——编者注

致谢

Neal 要感谢过去几年里在各种大会倾听过他演讲的所有人，他们帮助打磨和修订了本书。他也要感谢本书的技术审阅人员，他们为本书提供了非常宝贵的建议和意见，特别是 Venkat Subramanium、Eoin Woods、Simon Brown 和 Martin Fowler。Neal 还要感谢他的猫 Winston、Parker 和 Isabella，它们的干扰总能为他带来顿悟。他还想感谢他的朋友 John Drescher、ThoughtWorks 的所有同事、Norman Zapien 敏锐的耳朵、每年的 Pasty Geeks 度假团和附近的鸡尾酒俱乐部，感谢他们的支持和友谊。最后，Neal 要感谢他坚忍的妻子，她用笑容包容了他的经常出差和为事业所做的其他牺牲。

Rebecca 要感谢所有同事、大会出席者及演讲者和作者，感谢他们多年来为演进式架构领域贡献的想法、工具、方法以及提出的澄清问题。她也要感谢本书的技术审阅人员，感谢他们细致的阅读和评注。此外，Rebecca 还要感谢本书的合著者，感谢创作过程中所有启发性的对话和讨论。她要特别感谢 Neal，几年前与他有过一场关于应急式和演进式架构区别的大讨论或者说是辩论，之后这方面的思想才逐渐地成型。

Patrick 要感谢 ThoughtWorks 的所有同事和客户，感谢他们推动了需求，并为阐明演进式架构的构建想法提供了测试平台。他还要同 Neal 和 Rebecca 一起向本书的技术审阅人员表示感谢，他们的反馈极大地提升了本书的质量。最后，他要感谢本书的合著者们。过去几年中，他们位于不同的时区，相距甚远，很少有机会能面对面地一起工作，特别感谢能有机会在这个项目上和他们密切合作。

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章

软件架构

一直以来，由于软件架构涉及范围广且内涵不断变化，开发人员不断尝试给它一个简洁的定义。Ralph Johnson 就将其定义为“重要的东西（无论那是什么）”。架构师的工作就是理解和权衡那些“重要的东西”（无论它们是什么）。

为了给出解决方案，架构师工作的第一步是理解业务需求，也即领域需求。这些需求是使用软件来解决问题的动机，但终究只是架构师在构建架构时需要考虑的因素之一。架构师还必须考虑其他很多因素，其中一些比较明确（比如清楚地写在性能服务水平协议里），还有一些则隐含在商业活动中不言自明（比如公司正着手并购重组，软件架构显然也要有变动）。所以对于软件架构师来说，架构水平体现了他们在权衡业务需求和其他重要因素后找到最佳方案的能力。软件架构涵盖了所有这些架构因素，如图 1-1 所示。

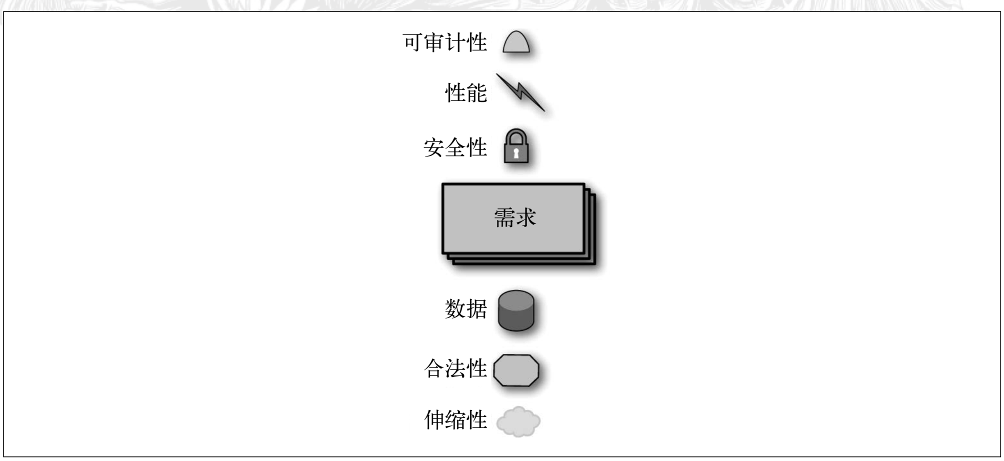


图 1-1：架构的完整概念，包括需求及“各种特征”

如图 1-1 所示，业务需求与其他架构关注点（由架构师定义）并存。这包括各种外部因素，这些因素可以改变构建软件系统的决策过程。表 1-1 列举了一些例子。

表1-1：部分特征列表

可访问性	可问责性	准确性	适应性	可管理性
可负担性	敏捷性	可审计性	自治性	可用性
兼容性	可组合性	可配置性	正确性	可信度
可定制性	可调试性	可降级性	可确定性	可演示性
可信赖性	可部署性	可发现性	分布性	耐久性
有效性	高效性	可用性	可扩展性	故障透明性
容错性	保真性	灵活性	可检查性	可安装性
完整性	互通性	可学习性	可维护性	可管理性
移动性	可修改性	模块性	可操作性	正交性
可移植性	精确性	可预测性	过程能力	可生产性
可证性	可恢复性	相关性	可靠性	可重复性
重现性	弹性	响应性	可复用性	稳健性
安全	伸缩性	无缝性	自我维持性	可服务性
安全性	简单性	稳定性	标准合规性	可生存性
可持续性	可裁剪性	可测试性	及时性	可追溯性

在构建软件时，架构师必须明确哪些特征最重要。然而，许多因素是互相矛盾的。比如，让软件具备高性能的同时还要实现极大的伸缩性就很困难，因为实现这两者需要谨慎地平衡架构、运维及其他诸多因素。因此，在为架构设计做必要分析的同时，又要处理好各个因素之间不可避免的冲突，架构师在权衡每个架构设计方案的利弊时，常常需要做出非常艰难的**折中**。近年来，软件开发核心工程实践的持续发展给我们提供了条件，使我们得以重新思考架构随时间的推移要如何变化，以及当这样的演进发生时，如何保护重要的架构特征。本书将这些部分联系起来，以一种新的方式思考**架构**和**时间**。

我们想为软件架构添加一个新的标准“特征”——**演进能力**。

1.1 演进式架构

无论我们怎么努力，软件依然变得越来越难以改变。由于各种原因，软件的组成部分不容易变更，而且随着时间推移变得愈发脆弱和难以操作。软件项目的变更通常是由于对功能或范围做了重新评估而导致的，但是还有一些变化是架构师和长期规划者无法控制的。尽管架构师喜欢为未来做战略性规划，但不断变化的软件开发环境使这一切变得困难重重。既然变化是必然的，那么我们就只能因势利导地来利用它。

1.1.1　一切都在变化，如何才能长期规划

生物世界中，环境因自然因素和人为因素而不断变化。例如，20 世纪 30 年代初，澳大利亚的甘蔗受到甲虫危害，导致甘蔗作物严重减产，利润大减。1935 年 6 月，作为应对措施，当时的甘蔗实验站管理总局引入了甘蔗蟾蜍来捕食甲虫，这种蟾蜍原本只产于中美洲和南美洲。短暂喂养后，1935 年 7 月和 8 月在昆士兰州北部投放了甘蔗蟾蜍。因为它们的皮肤有剧毒，并且在当地没有天敌，很快这种蟾蜍就泛滥成灾了。如今，澳大利亚的甘蔗蟾蜍大约有 2 亿只。这件事告诉我们：向高度动态的（生态）系统中引入变化，可能会产生无法预料的结果。

软件开发体系由所有的工具、框架、库以及最佳实践（软件开发领域的技术积累）构成。和生态系统一样，软件开发体系实现了平衡，开发人员能够理解这个体系并为其添砖加瓦。然而，这种平衡是动态的，随着新事物不断出现，平衡不断被打破和重建。想象一个脚踏独轮车，手里还拿着盒子的人。他是**动态的**，因为他需要不断调整来保持挺立；他又是**平衡的**，因为他保持着身体平衡。在软件开发体系中，每一项创新或新实践都可能打破现状，迫使系统重新建立平衡。就好比我们不断地将更多的盒子抛向骑独轮车的人，迫使他不断寻求新的平衡。

架构师在很多方面都和这个倒霉的独轮车手相似，不断地平衡以适应环境变化。持续交付这项工程实践使得这个平衡过程有了结构性的转变，它将过去孤立的功能（例如运维）合并到了软件开发的生命周期中，这让我们对**变化**的含义有了新的认识。企业级架构师不能再依赖静态的五年计划了，因为整个软件开发体系在不断变化，任何一个长期计划都可能变得毫无意义。

即便对经验丰富的实践者来说，颠覆性的创新也是难以预测的。比如，像 Docker 这样的容器化技术的崛起就是一个不可预知的行业转变。但我们仍然可以通过一系列小的演进找到一些蛛丝马迹。以前，操作系统、应用服务器和其他基础设施都是商品，需要斥巨资购买使用许可。当时许多架构设计着重于高效利用共享资源。渐渐地，Linux 变得足以支撑企业及应用，使得购买操作系统的**费用**降为零。接下来，通过 Puppet 和 Chef 等工具自动配置服务器的 DevOps 实践使得 Linux **运维工作**也不再需要成本。一旦开发环境免费并得到广泛应用，势必朝着更加通用和便携的方向发展，于是 Docker 应运而生。但如果没有之前所有的演进过程，容器化就不会发生。

我们所使用的编程平台也在持续演进。新的编程语言提供了更好的应用编程接口（API），提高了对新问题的灵活性和适用性。新的编程语言还提供了不同的范式和概念。例如，引入 Java 替代 C++，降低了编写网络代码的难度并改善了内存管理。回顾过去的 20 年，很多语言一直在持续改进它们的 API，与此同时，新的编程语言往往用于解决新的问题。编程语言的演变如图 1-2 所示。

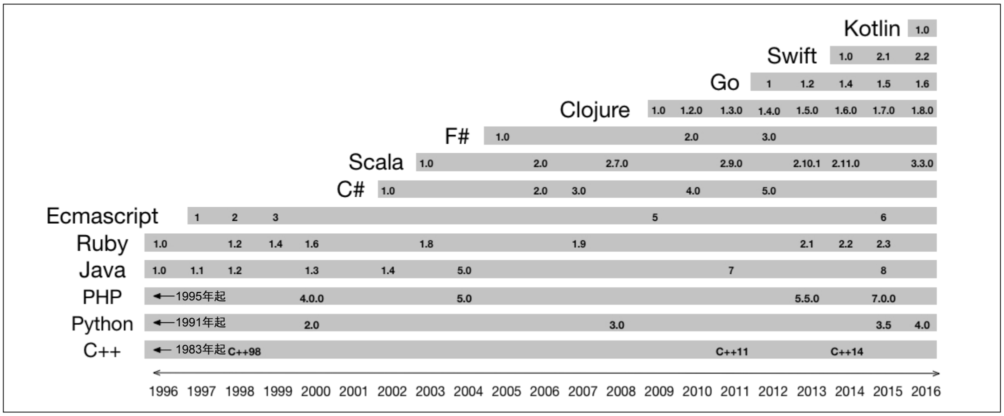


图 1-2：流行编程语言的演进过程

无论是在软件开发的哪个方面，比如编程平台、编程语言、运维环境、持久化技术等，我们都知道改变会持续发生。虽然无法预测技术或领域格局何时会改变，或哪些变化会持续下去，但我们清楚改变是不可避免的。因此，我们应该在构建系统的过程中对这一点保持清醒的认识。

如果整个体系持续地以出乎意料的方式发生变化，预测变化就变得不可能了，那么用什么来**替代**固定计划呢？企业架构师和开发人员必须学会适应变化。做长期计划有一个隐含的原因是财务上的考虑，因为以前软件变更的成本很高。但是，现代工程实践通过自动化和其他先进实践（例如 DevOps）降低了软件变更的成本。

多年来，一些聪明的开发人员发现系统的某些部分相对而言更难修改。这便是将**软件架构**定义为“将来难以变更的部分”的原因，这个省事的定义简单地区分了软件系统中真的难以修改的部分和可以轻松修改的部分。但这个定义依然不免走进了盲区，因为开发者预先假设“变更是困难的”，这变成了一个自证式的预言。

几年前，一些富有创新精神的架构师用新的视角审视了“将来难以变更”的问题。使架构具有可变性会怎样呢？换句话说，如果**易于改变**是架构的基本原则，那么变更将不再困难。反过来，使架构具备演进能力会导致一组全新的行为出现，进而再次打破整个体系的平衡。

即使环境不改变，架构特征出现磨损该怎么办？架构师设计出架构，将其置于纷乱的现实世界，基于架构执行各项事务。架构师要如何保护他们定义的重要部分呢？

1.1.2 完成架构构建后，如何防止它逐渐退化

有一种不幸的退化叫作架构**比特衰减**，它在很多组织中均有发生。架构师选择特定的架构模式来满足业务需求及让系统具备某些能力，但这些特征常常意外地随着时间推移而退

化。例如，架构师构建了一个包含顶部展现层、底部持久层和一些中间层的分层架构。负责报表功能的开发人员出于性能上的考虑经常会要求绕过中间层，直接从展现层访问持久层。架构师通过分层来隔离变化。而后开发人员绕过这些层，不仅增加了耦合，还使得分层变得毫无价值。

定义了那些重要的架构特征后，架构师如何**保护**这些特征不磨损呢？答案是添加**演进能力**作为新的架构特征，使其在系统演进时保护其他特征。比如，架构师追求架构设计的高伸缩性，但不希望在系统演进时削弱该特征。因此，**演进能力**是一种元特征和保护其他所有架构特征的架构封装器。

本书将阐明演进式架构的另一个作用，即它是一种为架构的重要特征提供保护的机制。我们探寻**持续架构**背后的理念。**持续架构**指构建架构的过程没有最终状态，它会随着软件开发体系的不断变化而演进，并保护重要的架构特征。我们不会尝试定义整个软件架构，因为已经存在很多定义了。我们通过引入**时间**和**变化**作为头等架构元素来扩展当前的定义。

我们对演进式架构的定义如下。

> 演进式架构支持跨多个维度的引导性增量变更。

1.2 增量变更

增量变更描述了软件架构的两个方面：如何增量地构建软件和如何部署软件。

在开发阶段，允许小的增量变更的架构更易于演进，因为对于开发者来说，变更范围相对更小。对部署而言，增量变更指业务功能的模块化和解耦水平，以及它们是如何映射到架构中去的。示例如下。

假设有一个小工具商家 PenultimateWidgets，该商家采用微服务架构和一些现代工程实践运营着一个产品目录页面。该页面的一个功能是让用户给出小工具的星级评分。PenultimateWidgets 的其他一些业务也需要评分，比如客服代表、物流服务商评价等，所以这些业务共享星级评分服务。某天，该服务的开发团队发布了新版本，新版本允许用户给出半星评价（一个小而重要的升级）。其他需要评分的服务不需要强制升级，而可以逐渐地在合适的时候迁移到新版本。PenultimateWidgets 的 DevOps 实践采用了架构级别的监控，不仅监控各个服务，还监控服务与服务之间的路由。当运营人员观察到在给定时间内没有请求路由至特定服务时，他们自动将该服务从体系中移除。

这是一个在架构级别进行增量变更的例子：如有需要，原服务和新服务可以并存。团队可以在适当（不太忙或必要）的时候完成迁移，当所有迁移都完成时，旧版本的服务将会作为垃圾被自动回收。

增量变更的成功需要一些持续交付实践的配合。并不是任何情况都需要所有这些实践，但通常它们会一起发生。第 3 章将讨论实现增量变更的方法。

1.3 引导性变更

一旦架构师选择了重要的架构特征，他们会把变更引导进入架构，以保护这些重要特征。为此，我们借用演化计算中的一个概念：**适应度函数**。该函数是一种目标函数，用于计算潜在的解决方案与既定目标的差距。在演化计算中，适应度函数决定一个算法是否在持续提升。换句话说，随着每个算法变体的产生，基于设计者对算法“适应度”的定义，适应度函数决定每个变体的“适应程度”。

对于演进式架构，随着架构的演进，我们有着类似的需求。我们需要评估机制，来评估变化对架构重要特征的影响，并防止这些特征随着时间的推移而退化。适应度函数的隐喻涵盖多种机制，包括度量、测试和其他检验工具。我们采用这些机制来确保架构不会以不良方式变更。当架构师确定了需要保护的架构特征时，他们会定义一个或多个适应度函数来提供保护。

以往，架构往往要划出一部分作为管理活动，最近架构师才接受了通过架构实现变更的思想。架构适应度函数允许在组织需求和业务功能的上下文中制定决策，并为明晰且可测试的决策奠定了基础。演进式架构并不是毫无约束或不负责任的软件开发方式。相反，它可以在高速变迁的业务、严谨的系统需求和架构特征间找到平衡。适应度函数驱动架构设计决策，并引导架构变更适应业务和技术环境的变化。

第 2 章将详细介绍使用**适应度函数**指导架构演进。

1.4 多个架构维度

> 不存在单独的系统。世界是一个整体。如何划分系统边界取决于讨论的主题。
>
> ——Donella H. Meadows

古希腊的物理学通过固定点分析宇宙，最终发展成了经典力学。但是到了 20 世纪初，随着仪器越来越精密，现象越来越复杂，人们逐渐从经典力学转向相对论。科学家意识到之前被视为孤立的现象其实是相互影响的。自 20 世纪 90 年代起，受到启发的架构师越来越多地将软件架构视作多维的。持续交付也将运维纳入了软件架构的范畴。软件架构师往往关注技术架构，但那只是软件项目的维度之一。如果架构师想构建可演进的架构，就必须考虑系统中所有会受变化影响的部分。正如物理学所讲的，万物都是相关联的，架构师深知软件项目是多维的。

为了构建可以不断演进的软件系统，架构师不能只考虑技术架构。例如，如果项目包含一

个关系型数据库，那么数据库实体之间的结构和关系也会随着时间的推移而不断变化。另外，架构师也不希望系统在演进过程中暴露安全漏洞。这些例子展示了架构的不同**维度**，它们通常在架构中以正交方式交织在一起。部分维度在常见的**架构关注点**范围之内（见表 1-1），但是架构**维度**的意义其实更广泛，它囊括了很多传统意义上技术架构范畴之外的东西。每个项目都有许多维度，架构师在考虑架构演进时必须要想到。下面是一些影响现代软件架构演进能力的常见维度。

- 技术

  架构中的实现部分：框架、依赖的库和实现语言。

- 数据

  数据库模式、表格布局、优化计划等。通常由数据库管理员（DBA）处理这类架构。

- 安全

  定义安全策略、指导方针和指定工具来帮助发现缺陷。

- 运维与系统

  关注架构如何映射到现有的物理或虚拟的技术设施中，包括服务器、机器集群、交换机、云等。

以上每个视角构成一个架构**维度**——为了支持特定视角而有意进行的划分。这里架构维度的概念涵盖了传统的架构特征和其他有助于构建软件系统的因素。随着问题和周遭环境的改变，我们想保护架构不磨损，每个维度都提供给我们一个审视架构的视角。

从概念上划分架构的方法有很多，比如 IEEE 的软件架构定义中的 4+1 视图模型。它关注不同角色的不同视角，将整个系统划分成了**逻辑视图**、**开发视图**、**进程视图**和**物理视图**。在著名的《软件系统架构》一书中，作者给出了软件架构的视点目录，它涵盖了更多角色。类似地，Simon Brown 的 C4 建模符号也从概念上帮助组织区分关注点。不过，本书没有尝试创建任何维度分类，而是识别那些现有项目中的维度。从实用角度来看，不论如何对关注点进行分类，架构师都需要保证这些维度不磨损。不同的项目有不同的关注点，这导致每个项目都有特定的维度。对于新项目，以上任何技术都能提供有用的见解，但是对于现有的项目，我们必须处理眼前的实际情况。

按照架构的维度思考，通过评估重要维度对变化的响应，架构师可以分析不同架构的演进能力。随着系统与互相冲突的问题（伸缩性、安全性、分布式、事务性等）关联得越来越紧密，架构师必须跟踪更多的维度。只有结合所有这些重要维度，思考系统将如何演进，才能构建出可以不断演进的系统。

项目的整个架构范围由软件需求和其他维度构成。当架构和整个体系随着时间的推移一起演进时，我们可以使用适应度函数来保护架构特征，如图 1-3 所示。

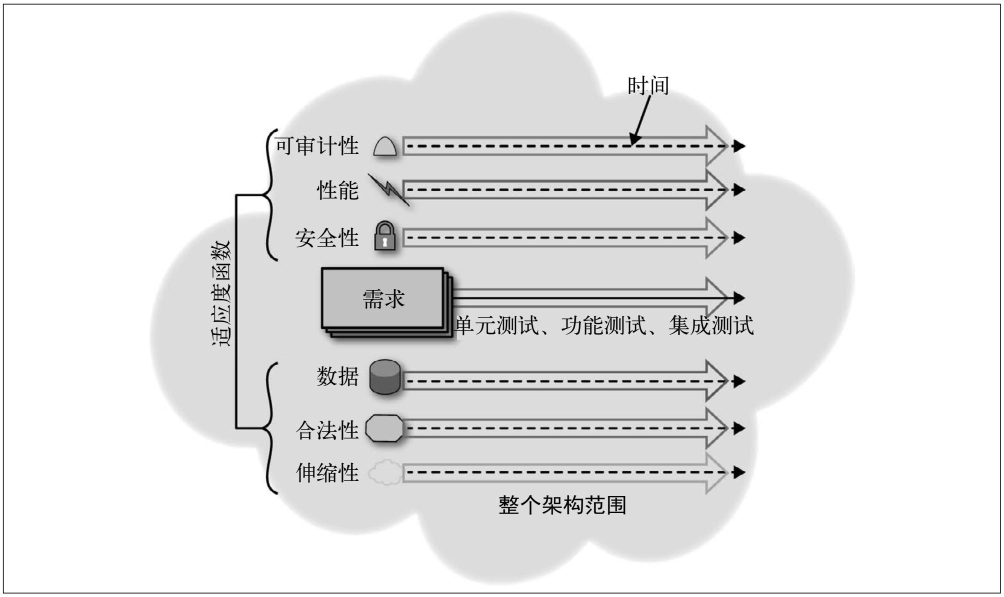


图 1-3：架构由需求和其他维度构成，每个维度都受适应度函数保护

在图 1-3 中，架构师确定了**可审计性**、**数据**、**安全性**、**性能**、**合法性**和**伸缩性**是该应用的关键架构特征。随着业务需求不断变化，每个架构特征都通过适应度函数来保护其完整性。

我们强调架构整体的重要性，但也应意识到，技术架构模式及相关议题也是架构演进的很大一部分，比如耦合和内聚。第 4 章将讨论技术架构耦合对演进能力的影响，第 5 章将讨论数据耦合的影响。

耦合不只和软件项目的结构元素有关。近来，很多软件公司认识到了团队结构对架构的影响。我们将讨论软件中的各类耦合因素，但是团队结构的影响出现得如此早且频繁，我们需要马上就此展开讨论。

1.5 康威定律

1968 年 4 月，梅尔文·康威在《哈佛商业评论》上发表了一篇名为“How Do Committees Invent?”的论文。在这篇论文中，康威提出：社会结构，特别是人与人之间的沟通途径，将不可避免地影响最终的产品设计。

康威描述道，在设计的最初阶段，人们首先需要高瞻远瞩地思考如何将职责划分为不同的模式。团队分解问题的方式会左右他们之后的选择，这便是**康威定律**。

> 在设计系统时，组织所交付的方案结构将不可避免地与其沟通结构一致。
>
> ——梅尔文·康威

正如康威所描述的，当技术人员将问题分解成更小的块，使其更易于委派时，就会产生协调问题。很多组织为了解决协调问题，会设置正式的沟通结构或是建立森严的等级制度，但这样的解决方案往往是僵化的。比如，按照技术职能（用户界面、业务逻辑等）划分的团队，他们在处理常见的跨职能问题时协调的开销就会增加。那些从创业团队跳槽到跨国公司工作的人，很可能会经历两种截然不同的文化：前者非常灵活且适应性很强，而后者的沟通结构往往是僵化的。康威定律一个很好的例子就是修改两个服务间的契约。如果一方想修改服务，而这需要另外一方的同意和配合，那么这件事就可能变得很难。

在论文中康威特别提醒软件架构师，不要只关注软件架构和设计，还应关注团队之间委派、分配和协调工作的方式。

在很多组织中，团队是根据职能来划分的。示例如下。

- 前端开发团队

  负责用户界面（UI）技术（如 HTML、移动端和桌面端）。

- 后端开发团队

  专门构建后端服务（有时还包括 API 层）。

- 数据库开发团队

  专门构建存储和逻辑服务。

在这样的组织中，管理层从人力资源的角度简单地按照职能划分团队，没有充分考虑工程效率。虽然每个团队都有其擅长的领域（比如构建一个视图，增加一个后端 API 或服务，或者开发一个新的存储机制），但是当需要发布新的业务功能或特性时，三个团队都要参与其中。各个团队通常都会针对眼前的任务优化效率，而不是针对那些更抽象的战略业务目标（特别是有工期压力时）。这会导致各团队往往专注于交付各自的组件，而不关注端到端的特性价值，导致这些组件可能无法高效协作。

在这样的团队编制下，由于每个团队都在不同的时间忙于自己的组件，因此那些依赖所有团队的特性需要花费更长的时间。例如，修改目录页这样常见的业务变更涉及 UI、业务规则和数据库模式的变更。如果每个团队都各自为战，那么他们必须协调时间表，这将增加实现该特性所需的时间。这个例子很好地解释了团队结构对架构和演进能力的影响。

康威在论文里提到："每当新的团队组建，其他团队的职责范围会缩小，能够有效执行的可选设计方案也会随之变少。"换句话说，人们很难改变其职责范围外的事情。软件架构师需要时刻关注团队的分工模式，从而使架构目标和团队结构保持一致。

很多构建架构（如微服务）的公司围绕服务边界构建团队，而不是按孤立的技术架构来划分。在 ThoughtWorks 技术雷达中，我们称之为"康威逆定律"。以这种方式组织团队是理

想的，因为团队结构会影响软件开发的很多维度，并且会反映问题的大小和范围。比如，在构建微服务架构时，企业通常会构建与架构相仿的团队结构，通过打破功能筒仓使每个团队都有人能考虑到业务的各个角度和技术的方方面面。

构建与目标系统架构相仿的团队结构，这样项目会更容易实现。

PenultimateWidgets 及其逆康威时刻

本书以 PenultimateWidgets 公司为例。它是排行倒数第二的小工具经销商，还是一家大型在线小工具（各种小商品）商家。这家公司正在更新其大部分 IT 基础设施，包括一些想暂时保留一段时间的遗留系统和正在迭代开发中的新战略系统。本书将重点介绍 PenultimateWidgets 所遇到的问题以及采取的解决方案。

他们的架构师首先观察了软件开发团队。旧的单体应用采用了分层架构，将展现、业务逻辑、持久化和运维分离开来。而他们的团队设置也和架构如出一辙：所有前端开发人员在一起工作，后端开发人员和数据库管理员也各自工作，运维则外包给了第三方。

当开发人员开始在架构（基于具有细粒度服务的微服务架构）上工作时，协调成本急剧增加。因为服务是围绕领域（例如**用户结算**）而不是技术架构来构建的，所以在变更单个领域时将需要各团队间进行大量的协调。

于是，PenultimateWidgets 采取了康威逆定律，建立了和服务范围相匹配的跨职能团队：每个服务团队包括服务负责人、几位开发人员、一位业务分析师、一位 DBA、一位质量保障人员和一位运维人员。

本书多处提到了团队的影响，并会通过示例说明它带来的结果。

1.6　为何演进

对于演进式架构，一个常见的问题是关于其名称的：为什么叫作演进式架构而不是别的？比如**增量式**、**持续式**、**敏捷式**、**响应式**或**应急式**，等等。这些术语都不够准确达意。这里所说的演进式架构包含了两个关键特征：增量和引导。

持续式、**敏捷式**和**应急式**都表达了随时间不断变化的概念，这确实是演进式架构的关键特征，但是这些术语都没能准确地表达出架构将如何变化，或者说期望的架构最终是什么样子的。虽然这些术语都隐含着环境变化，但是都没能涵盖架构应有的样子。而在演进式架构的定义中，**引导**的含义反映了我们想实现的架构，即我们的最终目标。

相比**可适应**这个词，我们倾向于**演进式**，因为我们对经历根本性演变的架构感兴趣，而不是那些经历了修修补补后变得越来越难以理解、复杂问题频发的架构。**适应**意味着解决方案不求优雅或长久，只要能找到方法使系统运作就行。为了构建能实实在在演进的架构，架构师必须支持真正的变化，而不是权宜之计的考虑。回到我们的生物学隐喻，演进是这样一个过程：建立一个适用的并能在其所处的不断变化的环境中持续运行的系统。系统可能存在个别的适应性，但是架构师应该关心整体可演进的系统。

1.7 小结

演进式架构主要由三方面构成：增量变化、适应度函数和适当的耦合。本书余下的部分将分别讨论它们，然后再将它们结合起来，帮助我们构建和维护支持不断变化的架构。

第2章

适应度函数

> 演进式架构是支持跨多个维度进行**引导性**增量变更的架构。
>
> ——本书定义

如前所述，**引导性**这个词表示架构应该朝着某个目标变更或者体现目标。本书借用了进化计算中的一个概念——适应度函数，遗传算法设计用它定义成功。进化计算有多种机制，通过对每一代软件进行小的变更使方案逐渐成形。工程师会评估每一代解决方案的当前状态，来判断当前方案与最终目标的距离。比如，在通过遗传算法优化机翼设计时，适应度函数会评估风阻、重量、气流，以及好的机翼设计应具备的其他特征。架构师通过适应度函数解释什么方案更好，并用它衡量何时能达到目标。在软件中，我们使用适应度函数检查开发人员是否保留了架构的重要特征。

我们对架构的适应度函数的定义如下。

> 架构的适应度函数为某些架构特征提供了客观的完整性评估。

适应度函数能够保护系统所需的各种架构特征。由于业务驱动、技术能力及其他诸多不同因素，系统和组织对架构的具体需求会有很大区别。有些系统要求很高的安全性，有些要求可观的吞吐量或低延迟，还有一些要求更好的故障恢复能力。这些对系统的考量形成了架构师所关心的架构特征。从概念上来讲，适应度函数体现了系统架构特征的保护机制。

我们也可以将**全系统适应度函数**看作适应度函数的集合，其中每个适应度函数对应架构的一个或多个维度。当适应度函数所对应的维度间存在冲突时，使用全系统适应度函数有助于我们做出必要的权衡。类似的问题在处理多功能优化时很常见，想同时优化所有值是不

可能的，因此我们必须做出选择。处理架构适应度函数时也一样，架构师熟知的经典案例是：由于加密的开销巨大，性能和安全性可能冲突，因此架构师必须做出艰难的权衡。架构师在调和对立力量时，很多时候为权衡犯难，比如在伸缩性和性能之间做出权衡。然而，对架构师而言，比较这些不同的特征是永恒的难题，因为它们从根本上是不同的（就好像苹果和橙子那样），并且所有利益相关者都认为自己所关心的特征最为重要。全系统适应度函数允许架构师通过统一的机制思考不同的问题，捕捉和保留重要的架构特征。图 2-1 展示了较小的适应度函数和它们所构成的全系统适应度函数之间的关系。

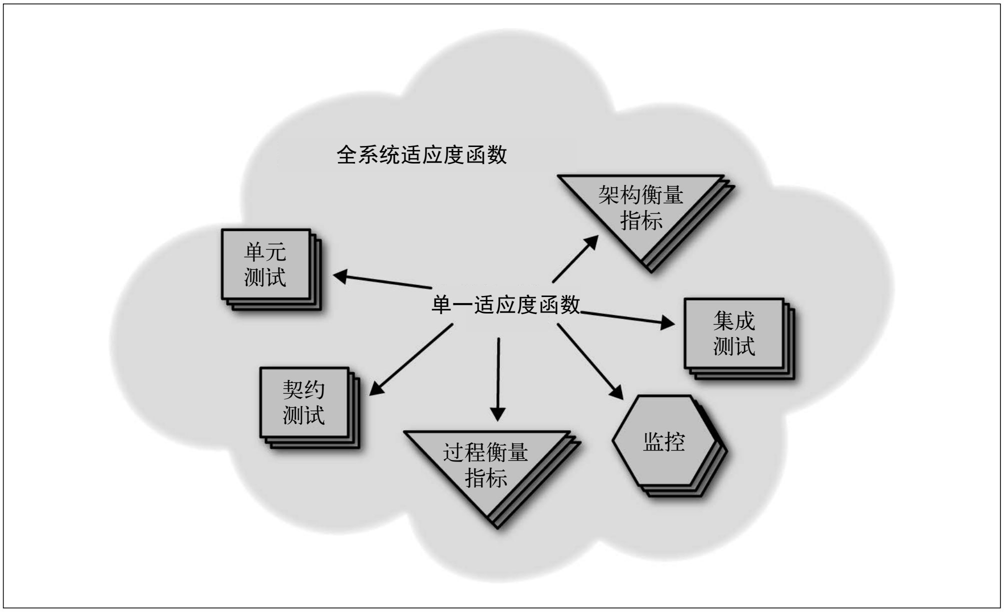


图 2-1：全系统适应度函数与单一适应度函数

全系统适应度函数对于架构演进至关重要，它为架构师提供了比较和评估不同架构特征的基础。与对待那些更具针对性的适应度函数不同，架构师很可能不会评估全系统适应度函数，尽管它为日后确定架构决策的优先级提供了指导。

> 系统绝不是其组成部分的总和，而是各部分相互作用的产物。
>
> ——Russel Ackoff 博士

没有指导原则，演进式架构便只是一个反动的架构。因此，对于任何系统，早期的关键架构决策都定义了重要的维度，如伸缩性、性能、安全性和数据模式等。从概念上讲，基于单个维度对系统全局行为的重要程度，架构师可以判断适应度函数的重要程度。

首先我们要更严格地定义适应度函数，然后再从概念上检验它们如何指导架构演进。

2.1 什么是适应度函数

数学上，函数从有效输入值集合中获得输入，将其转换为有效输出值集合中的唯一元素。在软件中，我们也普遍使用“函数”来指代可实现的东西。正如敏捷软件开发中的验收标准，适应度函数可能无法通过软件的方式实现（比如出于监管原因必须手动完成的某个过程），于是架构师还必须定义手动的适应度函数来指导系统演进。虽然我们倾向于实施自动化检查，但是有些项目无法自动化所有适应度函数。基于很多显而易见的原因，明确地阐明架构的验证方式并将其作为适应度函数对架构师依然有用。

正如第 1 章所述，现实架构由很多不同的维度构成，包括性能、可靠性、安全性、可操作性、代码规范和集成等方面的需求。我们希望适应度函数能代表每一项架构需求。开发人员有多种描述适应度函数的机制，比如测试和衡量指标。我们将通过几个例子多方面地了解不同类型的适应度函数。

利用适应度函数能很好地验证性能需求。考虑这样一个性能需求：所有服务调用必须在 100 毫秒内响应。我们可以执行一个测试（即适应度函数）来测量服务请求的响应时间，如果结果超过 100 毫秒则测试失败。为此，每个新服务的测试组件中都应该添加一个对应的性能测试。编写测试的开发人员必须决定范围的全面程度和输入的类型，才可能通过测试。他们还必须决定执行这些测试的时间以及应对测试失败的方法。团队应该尽早且频繁地测试性能，尤其在因代码变化造成性能突变（通常是往错误的方向上）的拐点。

我们也可以通过适应度函数保持代码规范。圈复杂度是常用的代码衡量指标，用来衡量函数或方法的复杂度。架构师可以设置一个阈值上限，然后在持续集成中运行单元测试来保护它，最后使用工具评估该衡量指标。在前面的例子里，架构师决定何时运行适应度函数来评估性能。对于代码规范而言，开发人员希望出现不规范的代码时构建立即停止，从而能积极地解决问题。

尽管很有必要，但是由于复杂性及其他约束，开发人员有时无法完整地执行所有适应度函数，比如对产生硬故障的数据库进行故障转移的时候。虽然自恢复的过程或许（也应该）是全自动的，但手动触发测试会更好。另外，尽管我们鼓励使用自动化脚本，但是手动确定测试成功与否或许更高效。

这些例子强调了适应度函数可以采用的不同形式，包括对其失败的快速响应，甚至开发人员应该在何时以及如何运行它们。虽然我们不一定仅运行一个脚本就获得“我们的架构目前综合适应度得分是 42”这样的结果，但我们能就与全系统适应度函数相关的架构状态进行严谨且明确的对话，也能讨论架构适应度可能引起的变化。

最终，所谓“适应度函数引导演进式架构”，指的是通过单独的适应度函数评估单个架构选择，同时通过全系统适应度函数确定变更的影响。适应度函数共同指出架构中对我们重要的部分，使我们能够在软件开发过程中做出各种关键又令人烦恼的权衡。

适应度函数将许多已有的概念统一为一个整体机制，让架构师可以统一思考许多现有的（往往是临时的）“非功能性需求”测试。收集重要的架构阈值和需求作为适应度函数，使得以前模糊又主观的评价标准变得更加具体。我们利用了大量现有机制来构建适应度函数，包括传统的测试、监控等工具。当然，并非所有测试都是适应度函数，只有当测试有助于验证架构问题的完整性时，它才是适应度函数。

2.2 适应度函数分类

适应度函数有很多不同的分类方式，可以依据其范围、运行频率、动态性及其他因素对其进行分类，必要时还可以对不同分类进行组合。

2.2.1 原子适应度函数与整体适应度函数

原子适应度函数针对单一的上下文执行，用来校验架构的某一维度，比如某个用来验证模块间耦合的单元测试（第 4 章将展示这类适应度函数的例子）。因此，一些应用级的测试被列入适应度函数的范畴，但并非所有单元测试都是，只有那些用于验证架构特性的才是适应度函数。

对于某些架构特征，开发人员不能只是孤立地测试各个架构维度。**整体**适应度函数在共享的上下文中运行，综合检验架构的多个维度，比如安全性和伸缩性。开发人员设计整体适应度函数来保证原子级特性能够正常地协同工作。比如，试想在架构中围绕安全性和伸缩性构建适应度函数，其中安全性适应度函数检查的关键是数据是否过时了，伸缩性适应度函数检查的关键是一定的延时内并发用户的数量。为了满足伸缩性的要求，开发人员执行缓存来使伸缩性适应度函数通过检查。当关闭缓存时，则安全性适应度函数通过检查。但当同时执行两个适应度函数时，缓存使得数据过于陈旧，导致系统无法通过安全性适应度函数的检查，整体测试失败。

显然，我们无法测试架构特征的所有组合，所以架构师需要使用整体适用度函数有选择性地测试那些重要的交互。在做出选择和确定优先级的过程中，架构师和开发人员会评估通过适应度函数实现特定测试场景的难度，从而评估该特征的价值。通常，架构关注点之间的交互决定架构的质量，而这正是整体适应度函数要解决的问题。

2.2.2 触发式适应度函数与持续式适应度函数

适应度函数间的另一个区别是执行频率。**触发式**适应度函数基于特定的事件执行，比如开发人员执行单元测试、部署流水线执行单元测试或质量保障人员执行探索性测试。触发式测试包含了传统测试，比如单元测试、功能性测试、行为驱动开发（BDD），还涉及其他测试开发人员。

持续式测试不是按计划执行，而是持续不断地验证架构的某些方面，比如事务处理速度。例如，在微服务架构中，架构师想对事务处理时间（完成单个事务平均花费的时间）构建适应度函数。任何形式的触发式测试都只能提供关于系统真实行为的少量信息。因此，开发人员不使用触发式测试，而是构建适应度函数，让它随着真实交易的发生来模拟生产环境中发生的事务。从而验证系统在实际使用时的行为，并收集真实的数据。

监控驱动开发（MDD）是另一种日益普及的测试技术。它通过监控生产环境来评估技术和业务的健康程度，而不是仅仅依赖测试。这些持续式适应度函数比标准的触发式测试更为动态。

2.2.3　静态适应度函数与动态适应度函数

静态适应度函数的结果是固定的，比如单元测试的二进制结果——成功或失败。该类型囊括了预定义期望值的适应度函数，比如二进制、数字区间、集合包含等。这类适应度函数通常会用到各种衡量指标。例如，架构师会为代码中的方法定义可接受的平均圈复杂度，并通过嵌在部署流水线的度量工具对其进行检查。

动态适应度函数依赖基于额外上下文变化的因素。某些值会视具体情况而定，比如在大规模运行的情况下，大多数架构师会采用较低的性能指标。例如，某公司可能基于伸缩性将性能指标设置为特定范围内的浮动值——在较大的规模下允许较低的性能。

2.2.4　自动适应度函数与手动适应度函数

显然，架构师喜欢自动化，自动化也是增量变更的一部分（第 3 章将深入研究）。因此，开发人员自然会选择在自动化环境中执行适应度函数，如持续集成、部署流水线等。事实上，在追求持续交付的过程中，开发人员使用 DevOps 完成大量工作，使得很多软件开发体系中曾被认为不可能的部分实现了自动化。这种有利的趋势需要持续下去。

然而，尽管我们希望软件开发中的每个部分都实现自动化，但某些部分却抗拒自动化。有时，系统的某些关键维度就无法自动化，例如对合法性的要求。在为某些问题域构建应用时，出于法律原因，开发人员必须手动认证来进行变更，这样的操作无法自动完成。类似地，某些团队可能希望项目变得更具演进性，但缺乏合适的工程实践。例如，在某些特定的项目上，大部分的质量保障工作仍然是手动的，并且这种情况在短期内不会改变。在以上两种（及其他）情况下，我们需要人为操作验证的**手动**适应度函数。

由此可见，虽然尽可能地消除手动步骤可以提高效率，但许多项目仍依赖必要的手动过程。我们需要为这些特征定义适应度函数，在部署流水线时添加手动阶段对其进行验证（第 3 章将详细介绍）。

2.2.5 临时适应度函数

虽然大多数适应度函数在变更发生时被触发，但架构师可能想通过时间组件评估适应度。例如，当项目使用了某个加密库时，架构师或许想创建一个临时适应度函数，在该加密库发生重大更新时发出提醒。**升级前破坏**（break upon upgrade）测试是这类适应度函数的另一种常见用法。例如，在某些平台的项目（比如 Ruby on Rails）中，一些开发人员不想等到下个版本发布时才能使用那些吸引人的新特性，于是他们将这些新特性直接移植到当前版本。但是移植过来的特性往往与实际发布的版本不兼容，因此开发人员通过**升级前破坏**测试来封装移植特性，迫使升级时重新对其进行评估。

2.2.6 预设式高于应急式

虽然在项目初期，架构师会定义大多数适应度函数，因为它们阐明了架构的特征，但有些适应度函数在系统开发阶段才显现。架构师无法在开始时就知晓架构的所有重要部分（第 6 章将讨论经典的**未知的未知问题**），因此必须随着系统发展不断确定适应度函数。

2.2.7 针对特定领域的适应度函数

某些架构有着特定的关注点，比如特殊的安全或监管需求。例如，一家处理跨国转账的公司可能设计特定的持续式整体适应度函数，模仿 Simian Army（第 3 章将介绍）对安全性进行压力测试。许多问题域都包含能引导架构师确定重要架构特征的因素。架构师和开发人员应该捕捉这些因素，并将其作为适应度函数来保证那些重要的架构特征不会随着时间出现磨损。

第 3 章的例子将结合这些维度来评估适应度函数。

2.3 尽早确定适应度函数

团队应该尽早确定适应度函数，将其作为初步理解全局架构关注点的一部分。团队还应该尽早确定**系统**适应度函数，来帮助他们确定想实现的变更。比较实现不同架构特征（及其适应度函数）的价值和难度，有助于更早地设置高风险工作的优先级，从而做出能够应对变化的设计。

没能确定适应度函数的团队将面临如下风险。

- 做出错误的设计选型，最终导致软件构建失败。
- 做出的设计选型在时间和成本上出现不必要的浪费。
- 系统无法轻松应对日后的环境变化。

对于任何软件系统，团队都应该尽早确定最重要的适应度函数及其优先次序。这有助于架

构师将大型系统拆解成更小的系统，使每个系统对应较少的适应度函数。

例如，有些企业处理一些安全敏感的数据，比如信用卡或支付详情。由于行业或工作的不同，存储这些信息意味着要遵循更严格的监管要求。由于相应的法律或标准的变化，抑或业务扩展到法规不同的州、地区或国家，这些监管要求可能随之改变。

一旦架构师确定了安全和支付在全系统适应度函数中的关键地位，那么团队可以设计出能综合考虑这些问题的架构。如果没有在早期确定适应度函数，团队可能将这些责任分散到了整个代码库中。那么当变更发生时，就需要更广泛地分析影响，导致变更总成本升高。

适应度函数可以简单分为三类。

- 关键维度

  这些维度对做出技术决策或设计决策至关重要。我们应该投入更多精力探索设计决策，使围绕这些维度的变更更容易。例如，对于银行应用，性能和回弹性是关键维度。

- 相关维度

  这些维度需要在特性级别考虑，但是不太会指导架构决策。例如，围绕代码质量的衡量指标很重要，但不是关键。

- 不相关的维度

  该类型的维度不会影响设计和技术决策。例如，像生产周期（从设计到实现所花费的时间）这样的流程衡量在某种意义上是重要的，但与架构无关，所以不必为其构建适应度函数。



将适应度函数的执行结果可视化至明显的公共区域，能使开发人员记得在日常编码中考虑它们，保持关键部分和相关适应度函数的活力。

对适应度函数进行分类有助于确定设计决策的优先级。如果一个设计决策对某个关键适应度函数有特定影响，那么应该花费更多时间和精力进行探针试验（时间可控的试验性编码工作）来校验设计的架构。有些团队采取基于集合的开发方式，它是精益和敏捷流程中的开发实践，用于同时设计多个解决方案。它以构建多套方案为代价来换取未来决策的可选方案。

2.4 审查适应度函数

适应度函数审查以会议的形式进行，会上主要业务和技术利益相关者会一起讨论如何修改适应度函数以满足设计目标。例如，当市场份额或用户数量显著增长时，或者引入新的功能或业务能力时，又或者大规模检修现有系统时，都必须审查适应度函数。

适应度函数审查大致涉及如下几点。

- 审查已有的适应度函数。
- 审查当前适应度函数的相关性。
- 确定每个适应度函数的规模或大小的变化。
- 确定是否有更好的方法测量或测试系统的适应度函数。
- 发现系统可能需要支持的新的适应度函数。

每年至少审查一次适应度函数。

PenultimateWidgets 和企业架构电子表格

当 PenultimateWidgets 的架构师决定构建一个新的项目平台时，他们首先创建了一个电子表格。该表格包含了所有期望的架构特征：伸缩性、安全性、回弹性和很多其他特征。但紧接着他们面临一个由来已久的问题：如果他们构建新架构来支持这些特征，那么如何确保它能持续提供支持？随着开发人员不断开发出新的功能，他们如何保护这些重要的架构特征不磨损呢？

解决方案是在表格中为每个关注点创建适应度函数，并重新制定其中一部分以便满足客观的评价标准。他们将适应度函数嵌入部署流水线（第 3 章将讨论）中对重要的标准进行持续验证，而不是偶尔为之。

虽然软件架构师有兴趣探索演进式架构，但我们并不想效仿生物演化过程。理论上，我们能构建一种可以任意突变并重新部署的架构。百万年后，它或许会进化成非常有趣的架构。但是，我们等待不了那么长的时间。

我们希望架构在引导下演进，所以我们在架构的不同方面设置约束来防止架构朝着错误的方向演进。例如犬类繁殖，通过选择想要的特征，我们可以在相对短的时间内培育出不同外形的狗。

下一章将介绍适应度函数的操作方面的更多内容。第 6 章会将适应度函数与其他所有架构维度结合起来。

第 3 章

实施增量变更

演进式架构支持跨维度进行引导式增量变更。

——本书定义

2010 年，Jez Humble 和 Dave Farley 发布了提高软件项目工程效率的一系列实践，即**持续交付**[1]。他们提供了通过自动化和工具来构建和发布软件的**机制**，但没有说明如何设计出可演进的软件。演进式架构以这些工程实践为前提，展示利用它们设计可演进软件的方法。

演进式架构的定义暗含了**增量变更**，这意味着这种架构更容易实现小的增量变更。本章会介绍支持增量变更的架构，以及实现增量变更的工程实践，这是演进式架构的重要构件。我们会讨论增量变更的两个方面：首先是**开发**方面，涵盖如何构建软件；然后是**运维**方面，涵盖如何部署软件。

下面是有关增量变更运维方面的一个例子。从第 1 章介绍的增量变更的例子开始，了解架构和环境部署的更多细节。小工具商家 PenultimateWidgets 有一个目录页面，该页面由微服务架构和相关工程实践提供技术支持，如图 3-1 所示。

注 1：Jez Humble 和 Dave Farley 合著的《持续交付：发布可靠软件的系统方法》已由人民邮电出版社图灵公司出版：ituring.com.cn/book/758。——编者注

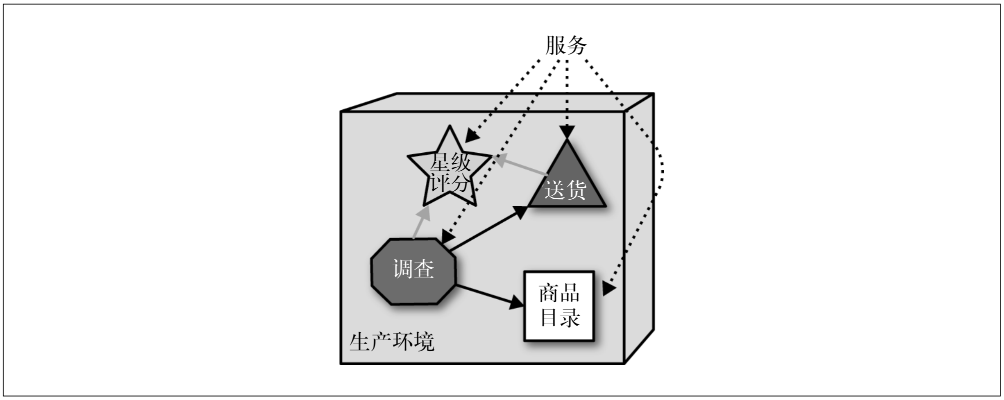


图 3-1：PenultimateWidgets 组件最初的部署配置

PenultimateWidgets 的架构师已经实现了可以独立运行的微服务。微服务是一种**无共享**架构，即每个服务都独立运行来消除技术耦合，因而能进行细粒度的变更。PenultimateWidgets 将所有服务分别部署到各自容器中来简化运维变更。

该网站允许用户对不同的小商品进行星级评分。而系统的其他部分也需要评分功能（如客服代表、物流服务商评价等），所以它们共享该星级评分服务。一天，星级评分服务团队发布了新版本。相比先前的版本，新版的重大更新是允许给出半星评分，如图 3-2 所示。

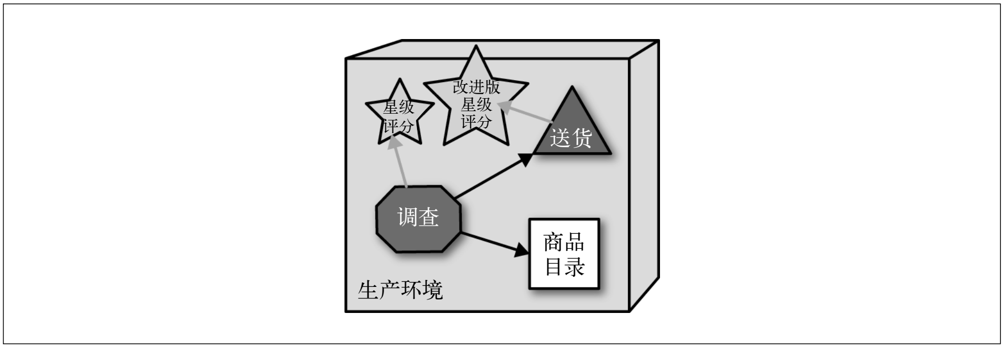


图 3-2：在部署了改进版星级评分后加入了半星评分

那些使用星级评分的服务不必迁移到新版的评分服务，而可以在方便的时候逐步转向更完善的服务。随着时间推移，系统中会有更多需要星级评分的部分转而使用改进后的版本。PenultimateWidgets 的 DevOps 实践采用了架构监控。它不仅监控服务本身，还监控服务间的路由。当运维团队观察到在一定时间内没有请求路由到某个服务时，将自动把该服务从整个生态系统中移除，如图 3-3 所示。

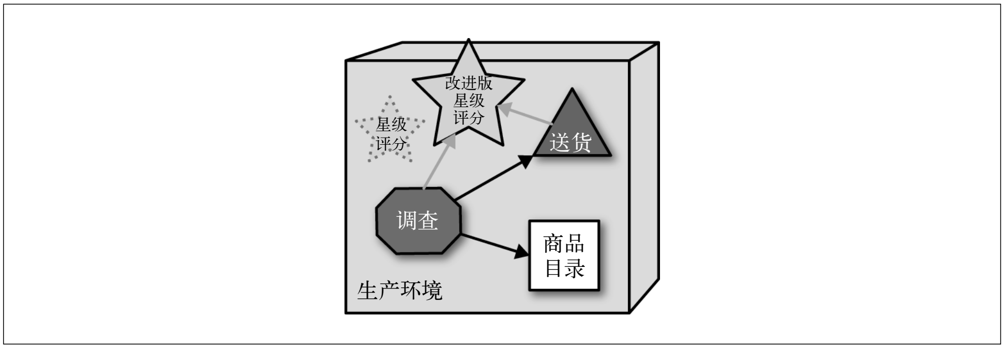


图 3-3：所有服务开始使用改进后的星级评分服务

演进能力是演进式架构的关键组成部分。我们从前面抽象的内容再深入一层。

PenultimateWidgets 构建了细粒度的微服务架构，每个服务使用容器技术（比如 Docker）进行部署，并通过服务模板处理基础设施的耦合。PenultimateWidgets 的应用包含了运行服务实例之间的路由（服务可包含多个实例，以便处理按需伸缩这样的运维问题）。架构师可以在生产环境中运行不同版本的服务，并通过路由控制访问。当部署流水线部署某个服务时，该服务将自己（服务的地址和服务契约）注册到服务发现工具中。当服务需要找到另一个服务时，它利用服务发现工具通过服务契约来获得目标服务的地址及适用的版本。

当部署新版星级评分服务时，它将自己注册到服务发现工具中并发布新契约。相比旧版本，新版服务增加了对半分值的支持。这样服务开发人员就不必担心限制所支持的值。如果新版要求调用者遵循不同的契约，通常会在服务内部处理，而不是让调用者判断应该调用哪个版本。6.5.10 节将介绍这种契约策略。

当团队部署新版服务时，他们不会强制要求调用服务立即升级到新版服务。因此，架构师可以暂时将星级评分端点添加到代理，通过代理检查请求发向哪个版本的服务，并将请求路由到所请求的版本。现有服务不需要任何修改，仍可继续使用旧版评分服务，但新的调用可以使用新版服务。旧版服务不会被迫升级，如需要仍可调用旧版服务。当调用服务决定采取新的行为时，他们修改从端点请求的版本。随着时间推移，原来的版本会被废弃，当不再需要旧版服务时，架构师便可以将其从端点删除。运维人员负责扫描那些不再被其他服务调用的服务（在一个合理的时间段内），并对其进行垃圾回收。

所有对该架构的变更（包括准备外部组件，比如数据库）都是在部署流水线的监督下进行的，使得 DevOps 无须协调部署中不相关的部分。

稍后将介绍构建支持增量变更的架构时所需的特征、工程实践、团队考量等内容。

3.1 构件

在持续交付及其工程实践的推动下，近年来很多能在架构级别提供灵活性的构件成为了主流。

软件架构师必须决定系统的构成方式，他们通常绘制不同规格的图表来完成此项工作。架构师常错误地将架构视为一个待解的**方程**。他们所购买的许多工具通过框、线和箭头使之更加明确。虽然这些图表能提供有用的二维视图（一个理想世界的快照），但我们身处于四维的世界。因此，为了扩展这个二维图表，需要为其添加更多细节。在图 3-4 中，架构师在真实的生产环境中使用软件验证他们的设计，`ORM` 标签变成了 `JDBC 2.1`，同时视图也进化成了三维。随着时间推移，业务和技术的不断变化要求架构师采用四维视图描绘架构，这使得演进成为重中之重。

软件中的一切都是动态的。以一台电脑为例，在电脑上安装操作系统和一套重要的软件后，把它放在柜子里锁上一年。当把电脑从柜子里拿出来，接通电源并连上互联网，你会看到它花很长的时间安装更新。可见，尽管电脑没有发生任何变化，但是**整个世界在不断变化**，这就是之前提到的动态平衡。因此，所有合理的架构都必须包含演进式的变更。

当我们知道如何将架构部署到生产环境，并按需对其进行升级来纳入不可避免的变更（如安全补丁、软件升级、架构演进等）时，我们便逐渐步入四维世界了。如图 3-4 所示，架构并不是一个静态的方程，而是一个持续过程的快照。

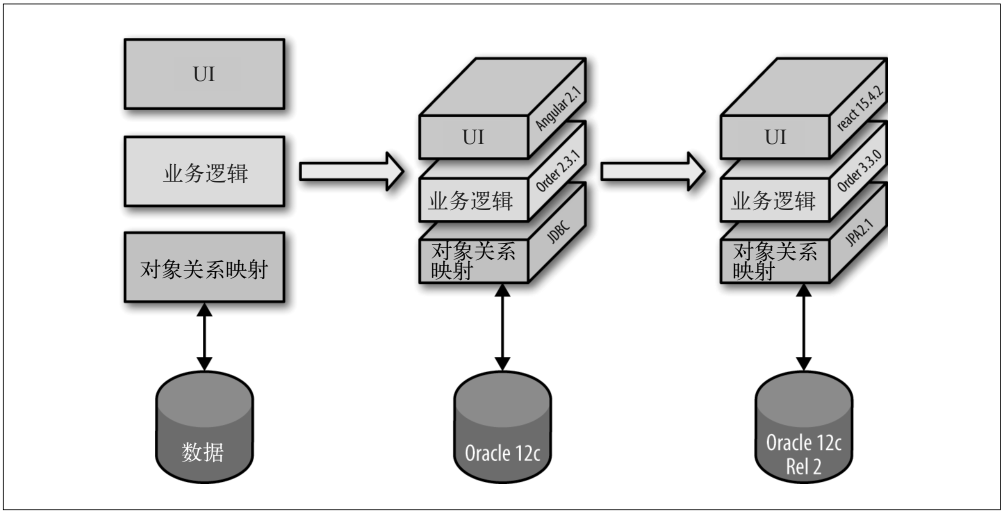


图 3-4：为了在现实世界中生存，现代架构必须是可部署和可变的

持续交付和 DevOps 的发展显示了实施架构并使其保持更新的必要性。进行架构建模并记录这些工作很好，但模型只是第一步。

架构在开始运行、具有生命力前是抽象的。

图 3-4 展示了在版本升级和新工具选择时，架构的自然演进过程。第 6 章将介绍架构的其他演进方式。

只有成功完成了架构**设计**、**实现**、**升级**和**无法避免的变更**后，甚至当架构能够经受由前期未知的未知因素引起的反常事件（第 6 章将介绍）带来的考验时，架构师才能评价架构的长期有效性。

3.1.1　可测试性

可测试性（架构特征是否能够通过自动化测试验证其正确性）是软件架构中一个被经常忽略的特性。但由于缺乏工具的支持，通常很难测试架构的各个部分。

但是，架构的某些方面确实可以轻松测试。例如，开发人员可以测试耦合、开发准则等具体的架构特征，并最终将这些测试自动化。

下面的例子展示了一个在技术架构维度中定义的适应度函数，用于控制组件之间耦合的方向性。在 Java 生态系统中，JDepend 是一个用于分析类包间耦合特征的度量工具。JDepend 由 Java 编写，开发人员可以在单元测试中调用其 API 来构建自己的分析工具。

示例 3-1 中的适应度函数，以 JUnit 测试的方式展示。

示例 3-1　通过 JDepend 测试验证 Java 包导入的方向性

```
public void testMatch() {
    DependencyConstraint constraint = new DependencyConstraint();

    JavaPackage persistence = constraint.addPackage("com.xyz.persistence");
    JavaPackage web = constraint.addPackage("com.xyz.web");
    JavaPackage util = constraint.addPackage("com.xyz.util");

    persistence.dependsUpon(util);
    web.dependsUpon(util);

    jdepend.analyze();

    assertEquals("Dependency mismatch",
             true, jdepend.dependencyMatch(constraint));
    }
```

示例 3-1 定义了应用中的包和导入规则。在基于组件的系统中，组件间的循环引用 / 依赖是一个令人头疼的问题（例如，当组件 A 引用了组件 B 时，组件 B 反过来也引用了组件 A）。

如果开发人员意外地在 `util` 包中导入了 `persistence` 包，那么这个单元测试将在代码提交前就失败。相比执行严格的开发准则（伴随着居高临下的说教），我们倾向于构建单元测试来捕捉架构违例。这样开发人员可以专注于领域问题而不是这些杂事。更重要的是，这使得架构师可以把规范整合为可执行的构件。

任何人都可以管理适应度函数，并且不同的团队和角色可以共同承担该职责。在示例 3-1 中，应用程序团队可能负责方向性适应度函数，因为这是该项目的特定关注点。在同一条部署流水线中，安全团队所负责的适应度函数可能横跨多个项目。通常不同的团队角色共同负责定义和维护适应度函数，其中包括架构师、开发人员和维护架构完整性的人员。

很多架构相关的事情都能被测试。可以用类似于 JDepend[2]（在 .NET 体系中类似的工具是 NDepend）的工具测试架构的结构特征。还可以用其他一些工具测试性能、伸缩性、回弹性及其他架构特征。任何帮助评估架构特征的工具都能用作适应度函数，监控和日志工具亦然。

一旦架构师确定了适应度函数，就应该确保及时地对其进行评估。自动化是持续评估的关键。**部署流水线**是进行此类评估的常用工具。使用部署流水线，架构师可以决定执行适应度函数的类别、时间和频率。

3.1.2　部署流水线

持续交付描述了部署流水线机制。和持续集成服务器类似，部署流水线在“监听”到变化后执行一系列验证步骤，每一步都更加复杂。持续交付实践鼓励使用部署流水线将常规项目任务自动化，例如测试、服务器准备和部署等。像 GoCD 这样的开源工具有助于构建部署流水线。

> **持续集成与部署流水线**
>
> 持续集成是敏捷项目中较出名的工程实践，它鼓励开发人员尽可能早且频繁地进行集成。为了便于持续集成，诞生了许多商业和开源工具，例如 ThoughtWorks CruiseControl。持续集成为团队提供了“正式”的构建场所，开发人员用它来保证代码能正常运行。同时，持续集成服务器还提供了执行日常项目任务的完美时间和场所。这些任务包括单元测试、代码覆盖率统计、软件度量、功能测试等。对很多项目而言，持续集成服务器包含一系列要执行的任务，所有任务都成功则表示构建成功。大型项目最终的任务列表会非常惊人。

注 2：ArchUnit 出现在最新的 ThoughtWorks 技术雷达上，可作为 JDepend 的替代品。

部署流水线鼓励开发人员将任务划分为不同的**阶段**。部署流水线包含了多阶段构建的概念，使得开发人员可以进行必要的后续检查。相比持续集成专注于集成，这种将任务划分为不同阶段的能力使得部署流水线的用途更为广泛，例如验证生产环境是否准备就绪。因此，部署流水线通常包括不同级别的应用测试、自动化环境准备和其他一系列验证职责。

有些开发人员尝试用持续集成来勉强代替部署流水线，但很快便发现它们无法对任务和反馈进行必要的拆分。

如图 3-5 所示，典型的部署流水线会自动构建部署环境（使用 Docker 容器或通过 Puppet 或 Chef 等工具构建的定制环境）。

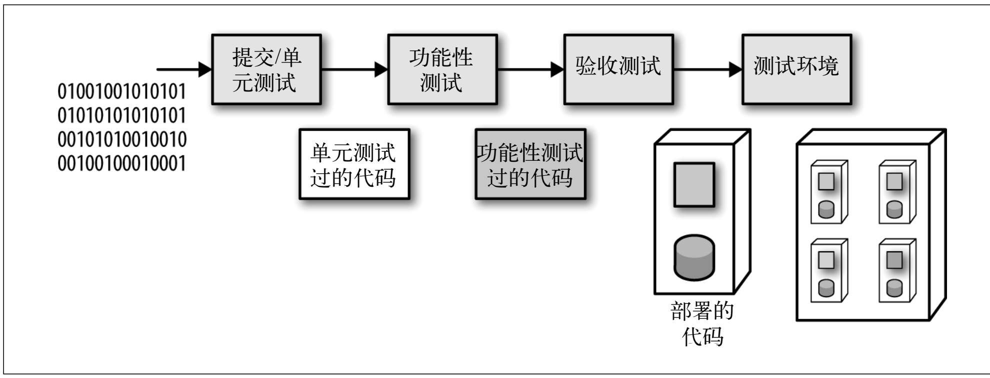


图 3-5：多阶段部署流水线

通过部署流水线构建部署镜像，使得开发人员和运维人员对部署更有信心。因为主机（或虚拟机）是声明式定义，并且通常从无到有地重建整个主机。

部署流水线还提供了执行架构适应度函数的理想方式——它适用于任何验证标准，在多个阶段包含不同抽象程度和复杂程度的测试，并在系统发生任何变更时执行这些测试。图 3-6 展示了添加了适应度函数的部署流水线。

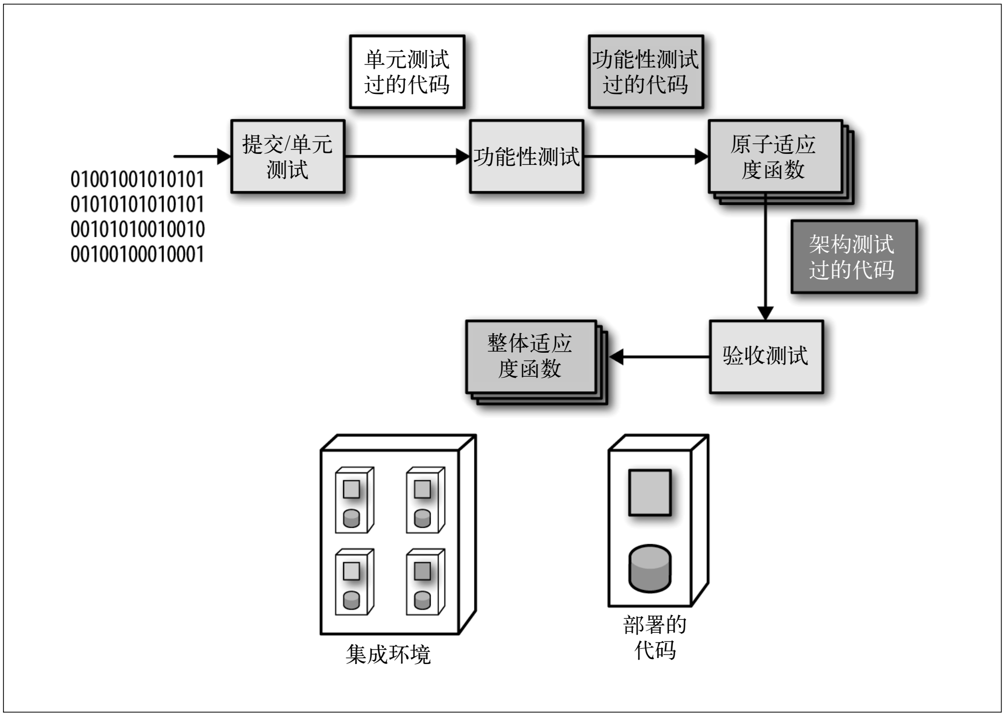


图 3-6：包含适应度函数阶段的部署流水线

图 3-6 展示了原子适应度函数和整体适应度函数，后者处于一个更复杂的集成环境中。部署流水线可以确保在系统每次变更时执行那些保护架构维度的规则。

PenultimateWidgets 的部署流水线

第 2 章介绍了 PenultimateWidgets 的需求电子表格。在采用了持续交付工程实践后，他们发现自动化部署流水线使得非功能性平台需求表现地更好。因此，服务团队构建了部署流水线来校验企业架构师和自己构建的适应度函数。现在，每当团队变更服务时，一系列测试将验证代码的正确性和服务在当前架构中的整体适应度。

在采用演进式架构的项目中，持续部署是另一种常见实践，即当变更通过了流水线测试及其他多重校验后，部署流水线将变更再部署到生产环境中。虽然持续部署是理想的工作流程，但它需要复杂的协调过程，开发人员需要保证持续部署到生产环境的变更不会造成任何破坏。

为了解决这类协调问题，通常在部署流水线中执行**扇出**操作，在流水线中并行执行多个任务，如图 3-7 所示。

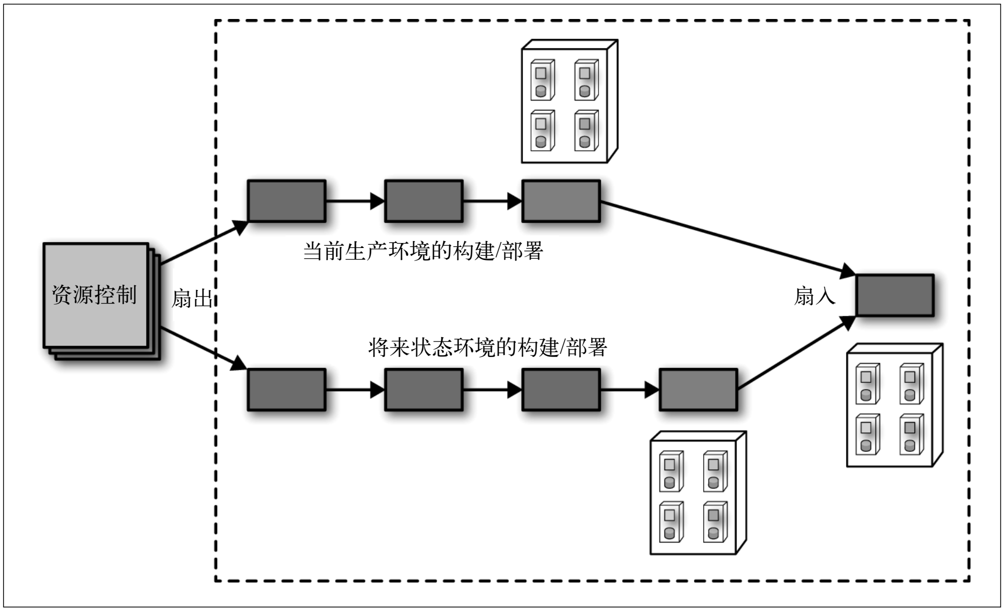


图 3-7：部署流水线扇出测试多个场景

如图 3-7 所示，当团队做出变更时，他们必须验证两件事：首先，变更没有对当前生产环境造成负面影响（因为部署流水线每成功运行一次都会将代码部署到生产环境）。其次，变更本身是成功的（影响未来的环境）。部署流水线的扇出操作允许任务（比如测试、部署等）以并行方式执行以节省时间。在完成一系列并行的任务后（如图 3-7 所示），流水线便能评估结果了。如果所有任务都成功了，那么流水线就可以执行扇入操作，将任务合并为单一操作线程来执行后续任务，例如部署任务。需要注意的是，当团队需要在不同上下文中评估变更时，部署流水线可能执行多次这样的扇出、扇入操作。

业务影响是持续部署带来的另一个常见问题。用户不希望经常出现大量新功能，而是希望以更传统的方式推出，比如“大爆炸”式的部署。最常用的方法是使用功能开关来同时实现持续部署和阶段性发布。通过实现隐藏在功能开关下的新功能，开发人员可以安全地将其发布到生产环境，而不用担心用户会过早看到它。

生产环境中的质量保障

开发新功能时使用功能开关使得团队能在生产环境中开展质量保障工作，这是使用功能开关的一个附带好处。很多公司没有意识到可以在生产环境进行探索性测试。一旦团队习惯于使用功能开关，便能将变更部署到生产环境，因为大部分功能开关框架支持开发者根据各种标准（如 IP 地址、访问控制列表等）来路由用户。如果在功能开关的控制下，团队部署的新功能只允许质量保障部门访问，那么他们就能在生产环境中进行测试。

在工程实践中使用部署流水线，架构师可以轻松地应用项目适应度函数。在设计部署流水线时，需要清楚哪些阶段对开发人员而言是最常见的挑战。将项目的架构关注点（包括演进能力）转换为适应度函数能带来很多好处。

- 适应度函数的结果客观且可量化。
- 捕捉所有关注点作为适应度函数，创造了一致的执行机制。
- 适应度函数列表便于开发人员设计部署流水线。

在项目的构建周期中确定执行适应度函数的时间和类别，以及找到适合的上下文并不容易。但是，一旦部署流水线中的适应度函数就位，架构师和开发人员就能确信演进式变更不会违反项目准则。架构关注点往往没有被充分阐明并且缺乏客观的评估，将它们定义为适应度函数可以使这个过程更加严谨，并提高对工程实践的信心。

3.1.3 组合不同类型的适应度函数

在部署流水线中执行适应度函数时，不同类型的适应度函数通常会发生交叉。下面介绍一些不同类型的适应度函数的常见搭配以及相应的示例。

- ❑ *原子适应度函数 + 触发式适应度函数*

  在软件开发过程中运行的单元测试和功能性测试就属于此类适应度函数。开发人员运行它们来验证变更，与此同时，自动化机制（例如部署流水线）通过持续集成保证验证的及时性。通过单元测试验证应用架构完整性的某些方面是此类适应度函数的常见例子，例如通过单元测试验证代码的循环依赖或圈复杂度。

- ❑ *整体适应度函数 + 触发式适应度函数*

  整体触发式适应度函数会作为集成测试的一部分在部署流水线中运行。开发人员设计它们专门来测试系统的各个方面如何按照定义好的方式进行交互。例如，开发人员可能想知道更严格的安全控制对系统伸缩性的影响。架构师设计它们来测试代码的一些集成特征，因为失败的测试能指出某些架构缺陷。正如所有触发式测试一样，开发人员通常在开发过程中、在部署流水线中或持续集成环境中同时执行这些适应度函数。通常这些测试和衡量会有明确的结果。

- ❑ *原子适应度函数 + 持续式适应度函数*

  持续测试作为架构的一部分运行，开发人员围绕这些测试开展设计工作。例如，架构师可能关心是否所有 REST 端点都使用了适当的动词、合理地处理错误并正确地支持元数据，为此构建了可持续运行的工具来调用这些 REST 端点（和普通客户端一样）并验证调用结果。这些适应度函数的**原子**范围表明它们只测试架构的某个方面，**持续性**则表现在这些测试作为系统整体的一部分运行。

- 整体适应度函数+持续式适应度函数

  全系统持续式适应度函数会不停地测试系统的多个部分。基本上，这种机制代表系统的一个代理（或客户端），它不断地评估架构和运维的质量。这类适应度函数在现实世界中的一个突出例子是 Netflix 的 Chaos Monkey。Netflix 在设计分布式架构系统时，在亚马逊云上运行其系统设计。但是工程师担心可能发生奇怪的事情，因为在云上他们无法直接控制系统的运行，比如高延迟、可用性和弹性等。为了消除忧虑，他们创建了 Chaos Monkey，并最终演化为完整的开源项目 Simian Army。Chaos Monkey“潜入”亚马逊数据中心，意想不到的事情发生了，比如延迟上升、可靠性降低，其他混乱也随之发生。由于 Chaos Monkey 的存在，每个团队构建的服务都必须具有回弹性。Conformity Monkey 是前面提到的 RESTful 验证工具，它按照架构师定义的最佳实践来检查每个服务。

需要注意的是，Chaos Monkey 并不是按计划运行的测试工具，它在 Netflix 的体系中持续运行。它不仅促使开发人员构建强大的系统，还不断地测试系统的有效性。将这种持续不断的验证内置于架构中，使得 Netflix 成为了世界上最强大的系统之一。Simian Army 是整体持续式适应度函数的一个典范。它同时针对架构的多个部分运行，很好地保持了架构特征（如回弹性、伸缩性等）。

对开发者来说，整体持续适应度函数最为复杂，但是可以发挥巨大作用，如下面的案例研究所示。

3.1.4 案例研究：在每天部署60次的情况下重建架构

GitHub 是著名的以开发者为中心的网站，有着激进的工程实践，平均每天部署 60 次。其博客“Move Fast and Fix Things”描述的问题让很多架构师心惊胆战。原来一直以来 GitHub 使用 shell 脚本执行 Git 命令行来处理合并，虽然能正常工作但无法很好地扩展。Git 的开发团队构建了名为“libgit2”的库来替代 Git 命令行所提供的功能，并在本地彻底测试了合并功能。

然后，他们必须把新方案部署到生产环境。该行为自 GitHub 初期完美运行至今。开发人员最后要做的事是在已有的功能中引入一些错误，而且他们还需要将这些错误作为技术债进行修复。幸运的是，GitHub 开发人员创造了开源框架 Scientist，它提供了整体持续测试来审查代码变更。代码示例 3-2 展示了 Scientist 的测试结构。

示例 3-2　创建 Scientist 试验

```
require "scientist"

class MyWidget
  include Scientist

  def allows?(user)
    science "widget-permissions" do |e|
      e.use { model.check_user(user).valid? } # old way
```

```
      e.try { user.can?(:read, model) } # new way
    end # returns the control value
  end
end
```

在代码示例 3-2 中，开发人员将现有行为封装到 use 块（也称**控制**）中，同时将试验性的行为放到 try 块中（也称**候选者**）。science 块在代码执行的过程中将处理以下细节。

- 决定是否运行 try 块

  开发人员配置 Scientist 来决定如何运行试验。例如在本例中，目标是更新 GitHub 的合并功能，其中 1% 的随机用户尝试了新的合并功能。无论哪种情况，Scientist 都会返回 use 块的运行结果，确保调用者在任何情况下始终能收到和现有行为相同的结果。

- 按随机顺序执行 use 块和 try 块

  Scientist 这样做是为了防止未知的依赖意外地掩盖漏洞。有时执行顺序或其他偶然因素会导致误判。通过随机执行，Scientist 降低了这些错误发生的可能性。

- 衡量所有行为的持续时间

  Scientist 的部分工作是进行 A/B 性能测试，所以 Scientist 内置了性能监控。事实上，开发人员可以零碎地利用该框架，比如，可以在没有性能试验的情况下使用 Scientist 衡量调用时长。

- 比较 try 块和 use 块的运行结果

  因为目标是重构已有行为，所以 Scientist 会比较并记录每次调用的结果来查看是否存在不同。

- 吞掉（但记录）try 块中发生的异常

  新的代码可能出现意外的异常。开发人员不希望最终用户看到这些错误，所以 Scientist 让这些异常对最终用户不可见（但会将错误记录下来用作分析）。

- 发布所有的信息

  Scientist 将所有数据以各种格式提供给外部访问。

在重构合并功能时，GitHub 的开发人员对 Scientist 进行了如下调用来测试新的实现（调用 create_merge_commit_rugged），如代码示例 3-3 所示。

示例 3-3　试验新的合并算法

```
def create_merge_commit(author, base, head, options = {})
  commit_message = options[:commit_message] || "Merge #{head} into #{base}"
  now = Time.current

  science "create_merge_commit" do |e|
    e.context :base => base.to_s, :head => head.to_s, :repo => repository.nwo
    e.use { create_merge_commit_git(author, now, base, head, commit_message) }
```

```
      e.try { create_merge_commit_rugged(author, now, base, head, commit_message) }
    end
  end
```

在代码示例 3-3 中，对 `create_merge_commit_rugged` 的调用占调用总数的 1%，但正如示例所示，就 GitHub 的规模而言，所有极端情况都会很快被覆盖。

当执行代码时，最终用户总是收到正确的结果。如果 `try` 块返回不一样的结果，那么会被记录下来，随后 `use` 块的结果返回给最终用户。因此，对最终用户而言，最坏的情况不过是得到重构前合并功能的执行结果。如果在试验进行 4 天后，并且在最近 24 小时内没有出现缓慢的情况或不匹配的结果，那么 GitHub 的开发人员将移除旧代码并保留新代码。

在我们看来，Scientist 就是一个适应度函数。该案例突出展示了整体持续式适应度函数的战略用途，它让开发人员对重构系统的关键部分更有信心。通过运行新版本和现有版本，他们修改了架构的关键部分，并从根本上将遗留实现变为一致性测试。

大多数架构中通常含有大量的原子适应度函数和几个关键的整体适应度函数。最根本的决定性因素可归结为开发人员的测试目标以及测试结果的广度。

3.1.5 目标冲突

敏捷软件开发流程告诉我们：开发人员越早发现问题，那么解决问题的成本将越低。考虑所有架构维度的一个副作用是早期针对不同维度的目标会互相冲突。例如开发人员所在的组织后续想追求最激进的变更频率来支持新功能。代码的快速变更意味着数据库结构的快速变更，但 DBA 更关心稳定性，因为他们构建的是数据仓库。这两个演进目标在技术架构和数据架构间相互冲突。

显然，考虑到影响根本业务的诸多因素，团队必须做出一些妥协。使用架构维度识别部分架构关注点（并通过适应度函数对其进行评估），让我们能对不同的关注点进行“苹果和苹果”（同类事物）的比较，使优先级更加明确。

目标冲突无法避免。但是，尽早发现和量化这些冲突可以使架构师做出更明智的决定，制定出更清晰的目标和原则。

3.1.6 案例研究：为PenultimateWidgets的发票服务添加适应度函数

示例公司 PenultimateWidgets 的架构包含一个处理发票的服务。负责该服务的团队想替换掉过时的库和方案，但不希望这些变更影响其他团队与他们的集成。

发票服务团队定义了如下需求。

- 伸缩性

  虽然性能不是 PenultimateWidgets 关注的重点，但是他们为多个经销商处理发票细节，所以发票服务的可用性必须满足服务水平协议的要求。

- 和其他系统的集成

  PenultimateWidgets 体系中的其他几个服务也使用发票服务。团队希望进行内部变更时不会破坏这些集成点。

- 安全

  发票意味着钱，所以安全是始终是要考虑的问题。

- 可审计性

  一些州的法规要求，税收相关代码的变更必须通过独立会计师的审计。

发票服务团队使用了持续集成服务器，运行代码的环境最近升级为可按需配置。为了实现演进式架构适应度函数，他们构建了部署流水线来代替之前的持续集成服务器，并在流水线中创建了几个执行阶段，如图 3-8 所示。

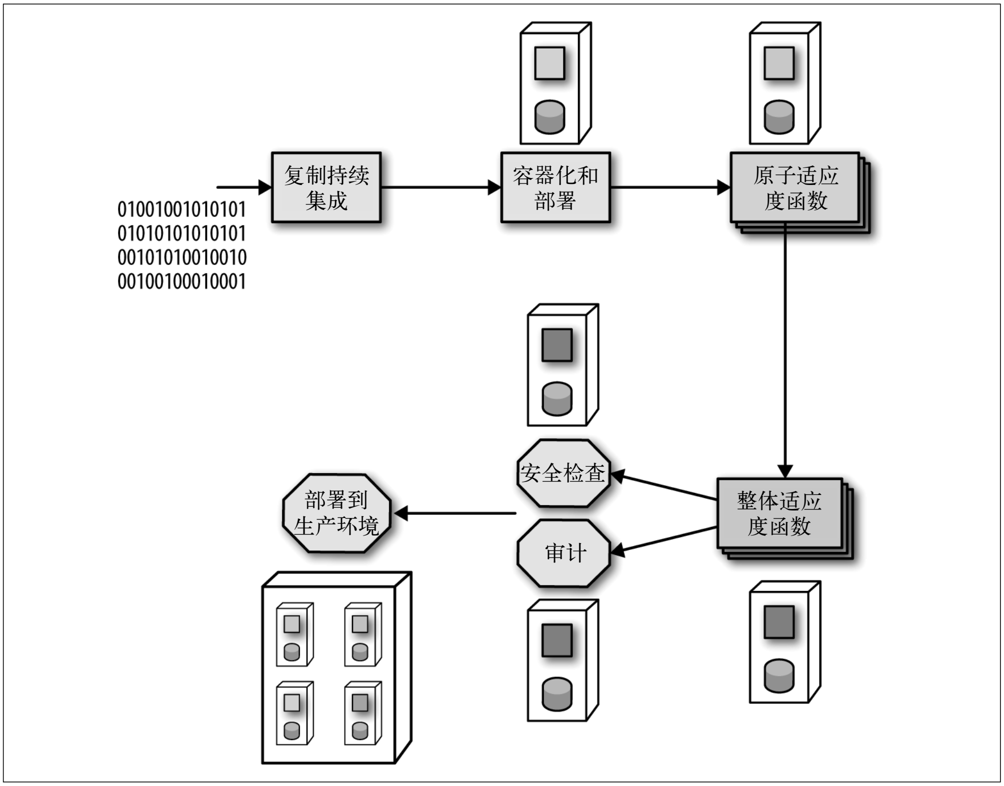


图 3-8：PenultimateWidgets 部署流水线

PenultimateWidgets 的部署流水线由六个阶段构成。

- 第一阶段——复制持续集成

  第一阶段复制了之前持续集成服务器的行为，运行单元测试和功能测试。

- 第二阶段——容器化和部署

  在第二阶段，开发人员为其服务构建容器，从而实现更深层次的测试，包括将容器部署到动态创建的测试环境中。

- 第三阶段——原子适应度函数

  在第三阶段原子适应度函数中，执行自动化的伸缩性测试和安全渗透测试。该阶段还会运行度量工具来标记在特定包中被开发人员修改过的代码，涉及可审计性。虽然该工具不做任何决定，但它会帮助后续阶段将覆盖范围缩小到特定的代码。

- 第四阶段——整体适应度函数

  第四阶段的重点是整体适应度函数，包括运行契约测试来保护集成点和一些进一步的伸缩性测试。

- 第五阶段 a——手动安全检查

  这是一个手动阶段，组织内特定安全团队将手动检查、审计和评估代码中的安全漏洞。部署流水线允许定义手动阶段，由相关的安全专员根据需要来触发。

- 第五阶段 b——手动审计

  PenultimateWidgets 位于斯普林菲尔德，该市所在的州对审计规则有具体要求。发票服务团队在部署流水线中构建了手动审计阶段，提供了一些便利。首先，将审计作为适应度函数让开发人员、架构师、审计人员及其他人员能以统一的方式思考这个行为，即通过必要的评估确定系统功能正确与否。其次，将评估过程添加到部署流水线，使得开发人员可以评估该行为的工程影响，使之与部署流水线中的其他自动化评估一样。

  例如，如果安全检查每周进行一次，而审计每月进行一次，显然，阻碍更快发布的瓶颈就是审计阶段。将安全检查和审计作为部署流水线中不同的阶段，可以更合理地做出与两者相关的决定。例如对公司而言，为了实现更频繁的发布，让审计顾问更频繁地进行必要的审计是否值得？

- 第六阶段——部署

  最后一个阶段是将变更部署到生产环境。在 PenultimateWidgets，这是一个自动阶段，只有当上游的两个手动阶段（**安全检查**和**审计**）都成功时才会被触发。

PenultimateWidgets 的架构师每周会收到一封自动生成的报告。报告包含了适应度函数的成功 / 失败率，它帮助架构师衡量适应度函数的健康状况、执行频率及其他情况。

3.2 假设驱动开发和数据驱动开发

3.14 节中 GitHub 对 Scientist 框架的使用就是一个**数据驱动开发**的例子——使用数据来驱动变更，并集中精力于技术变更。另一个类似的方法是**假设驱动开发**，该方法更关注业务问题而非技术问题。

在 2013 年圣诞节到 2014 年新年的那一周，Facebook 遇到一个问题：那一周内上传到 Facebook 的图片比 Flickr 上所有的图片还多，并且有 100 多万张图片被标记为令人不适的。Facebook 允许用户标记他们认为令人不适的图片，然后 Facebook 会审查这些图片来判断是否确实如此。但是图片数量的猛增产生了一个问题——没有足够的人手来审查这些图片。

幸运的是，Facebook 的现代 DevOps 能力使得他们能够对用户进行试验。当问到一个典型的 Facebook 用户有多大概率参与试验时，Facebook 工程师的回答是："100%，因为通常我们会同时进行 20 多个试验。"他们利用这种试验向用户提出问题，来了解为什么用户认为这些图片令人不适，并由此发现了人类行为的很多有趣的怪癖。例如，人们不愿承认他们不上相，但是他们会坦率地承认摄影师的摄影技术很差。通过不同的措辞和问题，工程师可以询问真实用户，并确定为什么他们把图片标记为令人不适的。在相对较短的时间内通过构建允许进行试验的平台，Facebook 剔除了足够多的误判，有效地管理了令人不适的图片。

在《精益企业》这本书中，Barry O'Reilly 介绍了**假设驱动开发**的现代化过程。在这个过程中，团队应该利用科学手段，而不是收集正式的需求然后花费时间和资源将功能构建到系统中。一旦团队创建出应用的最小可行产品（无论是新产品还是维护现有产品），他们便能在构思新功能时建立假设，而不是需求。假设驱动开发的假设是根据假设来检验的，什么试验可以确定结果以及用什么验证假设意味着应用开发的走向。

例如，与其在目录页面修改促销商品的图片尺寸（因为业务分析师认为这是个好主意），不如做出假设：假设促销商品图片大一点，那么这些商品的销量会增加 5%。一旦做出假设，便可以通过 A/B 测试（一组的促销图片更大，另一组不然）来进行试验并计算结果。

随着业务用户不断增加，即使是敏捷项目也会变得糟糕。业务分析师的个人决定单看上去也许合理，但与其他功能结合时，最终可能降低了整体体验。在下面这个出色的案例研究中，mobile.de 遵循了一个逻辑——随意发布新功能导致销量下滑，至少一定程度上是因为 UI 变得太复杂，这通常是在成熟的软件产品上持续开发的结果。几种不同的解决思路有：更多列表、更好的排序或更好的分组。为了做出决定，他们构建了三个不同版本的 UI 让用户来决定。

驱动敏捷软件方法论的引擎是内置的反馈环，如测试、持续集成和迭代等。然而包含应用程序最终用户的反馈环已经脱离了团队的控制。使用假设驱动开发，我们能以一种前所未有的方式将最终用户纳入构建流程，从他们的行为中学习并构建出对其真正有价值的系统。

假设驱动开发需要协调很多动态的部分，包括演进式架构、现代 DevOps、变更需求收集和同时运行多版本应用的能力。基于服务的架构（比如微服务）通常通过服务间的智能路由实现不同版本服务的并行。例如，某个用户可能通过某个特定的服务组访问应用，而另一个请求可能访问相同服务中完全不同的一组实例。如果多数服务都包含多个运行实例（例如为了伸缩性），那么让一部分实例运行增强过的功能，并将部分用户路由到这些功能会变得很简单。

为了产生可观的结果，试验应该进行足够长的时间。通常最好找到某种可衡量的方式来确定更好的结果，而不是通过弹出窗口等形式的调查来打扰客户。例如，某个假设的工作流能否让客户用更少的键盘输入和点击完成任务？在不打扰用户的情况下将其纳入开发和设计的反馈环中，可以构建出更实用的软件。

3.3 案例研究：移植什么

PenultimateWidgets 的某个应用承担主要工作，它是在过去十年的大部分时间里开发的 Java Swing 应用，并在不断地添加新功能。公司决定将它移植到网页应用中。然而，业务分析师面临艰难的抉择——应该从现有的繁多的功能中移植多少呢？具体而言，为了能够尽快交付最重要的功能，他们应该以什么顺序将功能移植到新应用呢？

PenultimateWidgets 的某个架构师问业务分析师“最受欢迎的功能有哪些”，业务分析师竟然不知道！尽管多年来他们一直在详细说明该应用的细节，但他们并不知道用户是如何使用应用的。为了了解用户，开发人员发布了新版的遗留应用，在其中添加了日志功能来追踪用户实际使用了哪些菜单功能。

几周后，他们得到了结果，绘制出一份的线路图，确定了移植相关功能的顺序。他们发现发票和查找客户功能最为常用。意外的是，该应用中他们花费了很大精力构建的一小部分几乎没有用处，于是团队决定不把它们移植到新的网页应用中。

第 4 章

架构耦合

关于架构的讨论常常会归结到耦合——架构各个部分是如何连接并相互依赖的。很多架构师认为耦合是必然之恶，但是如果不依赖其他组件（并与其耦合）又很难构建复杂的软件。演进式架构注重适当的耦合，即如何确定哪些架构维度间应该相互耦合来以最小的开销和成本最大程度地获益。

4.1 模块化

首先，我们厘清一些常见术语，在架构讨论中它们常被提到甚至滥用。平台不同，代码复用机制也不同，但它们都支持将相关代码组成**模块**。**模块化**描述了相关代码的逻辑分组。可以以不同的物理方式封装模块。**组件**就是模块的物理封装。模块意味着**逻辑**分组，而组件意味着**物理**划分。

开发人员发现进一步拆分组件有助于一些工程实践，例如构建和部署。**库**是其中一类组件，它往往和调用代码在相同的内存地址内运行，通过编程语言的函数调用机制进行通信。库常用作编译时的依赖。由于大多数复杂应用是由各式各样的组件构成的，因此在应用程序架构中存在很多和库相关的问题。另一类组件被称为“服务”，倾向于在自己的地址空间中运行，通过低级网络协议（比如 TCP/IP）、更高级的网络协议（比如简单对象访问协议，SOAP），或表述性状态转移（REST）进行通信。服务相关问题往往在集成架构中出现，因为它造成了运行时的依赖。

所有模块机制都有助于代码复用，在任何级别尝试复用代码都是明智的选择，无论是单一的函数，还是封装好的业务平台。

4.2 架构的量子和粒度

软件系统以各种方式相互联接。软件架构师通过许多不同的视角分析软件。但是组件级的耦合并不是联接软件的唯一方式。许多业务概念在语义上联接系统的各个部分，这便产生了**功能内聚**。要想使软件成功地演进，开发人员必须考虑所有可打破的耦合点。

正如物理学所定义的，**量子**是物理实体相互作用时所涉及的最小单位。**架构量子**则是具有高功能内聚并可以独立部署的组件，它包括了支持系统正常工作的所有结构性元素。在单体架构中，量子就是整个应用程序，每个部分都高度耦合，因此开发人员必须对其进行整体部署。

> **领域驱动建模中的限界上下文**
>
> Eric Evans 所著的《领域驱动设计》对现代架构思想影响深远。领域驱动设计（DDD）是一种建模技术，运用该技术可以有组织地分解复杂的问题领域。DDD 定义了**限界上下文**，所有领域相关内容在该领域内是可见的，但不对其他限界上下文可见。在 DDD 出现之前，开发者在组织内部寻求可全局复用的通用实体。然而，创建通用的共享组件导致了一系列问题，比如耦合、协调难度和复杂度增加。限界上下文的概念指出，各个实体在其本地具体上下文中表现最佳。因此，每个领域应创建自己的实体，并在集成点协调差异，而不是在整个组织中创建统一的 `Customer` 类。DDD 影响了多种现代架构和一些相关因素，例如团队结构（见 1.5 节的示例）。

相比之下，微服务架构在架构元素之间定义了物理限界上下文，封装了所有可能变化的部分。这种架构就是为了增量变更而设计的。在微服务架构中，限界上下文作为量子边界，包含了服务所依赖的组件，比如数据库服务器。它还包含一些架构组件，例如搜索引擎、报表工具及任何有助于交付功能的组件，如下图所示。

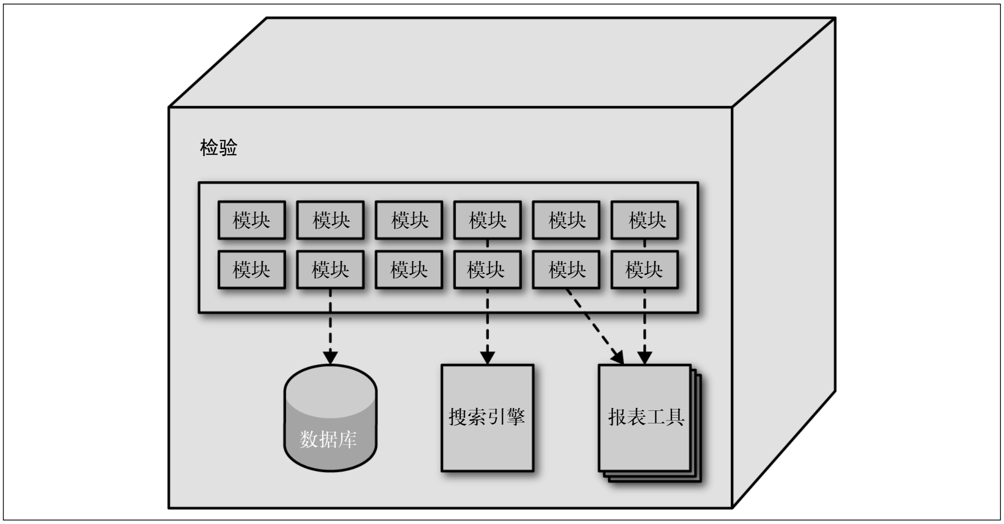


图 4-1：微服务中的架构量子包含了服务本身和它所依赖的各个部分

在图 4-1 中，该服务包含了代码组件、数据库服务器和一个搜索引擎组件。微服务中的限界上下文理念使服务的各个部分在一起运行，严重依赖现代 DevOps 实践。下面探究一些常见的架构模式及其典型量子边界。

在演进式架构中，传统上独立的角色间需要协作，比如架构师和运维人员。架构在真正运转前是抽象的，开发人员必须了解组件在实践中如何协同工作。无论开发人员选择了哪种架构模式，架构师都应该显式定义架构量子的大小。小的架构量子意味着更快的变更速度，因为其影响范围更小。通常小组件比大组件更易于使用。量子的大小决定了架构中进行增量变更的可能性（量子越小，可能性越大）。

在物理学中，自然界存在 4 大基本相互作用力：**引力**、**电磁力**、**强力**和**弱力**。强核力的力量不容小觑，它将原子（普通物质）聚集在一起。将其打破会因核裂变而释放巨大的能量。类似地，我们很难将一些架构组件分解成更小的部分，就好像它们也具有强核力一样。构建演进式架构的关键之一在于决定自然组件的粒度以及它们之间的耦合，以此来适应那些通过软件架构支持的能力。

在演进式架构中，架构师面对的是**架构量子**，系统中难以打破的力量将其聚集在一起。例如，事务就像强核力一样将一些不相关的部分绑定在一起。虽然开发人员可以分离事务上下文，但是过程很复杂，并且经常导致复杂情况发生，比如分布式事务。类似地，业务各部分可能高度耦合，因此将这种应用分解成更小的架构组件也许并不可取。

图 4-2 总结了它们之间的关系。

庞大的列表

我们从事的项目多年来以汽车拍卖为核心业务。不意外的是，系统中的 Listing（列表）类非常庞大，如同一个怪兽。为了解决它引起的协调问题，开发人员进行了几次技术重构试图分解这个庞然大物。最终的方案是将其中的关键部分 Vendor（供应商）分离到一个单独的类中。虽然技术重构成功了，但是开发人员和业务分析师之间出现了问题。开发人员不断地修改 Vendor，但是业务分析师并不将 Vendor 看作单独的实体。开发人员在项目上违反了 Eric Evans 在 DDD 中所提到的**通用语言**——确保团队中所有术语的意义相同。虽然这样能使开发人员更方便地拆分功能，但是它违反了业务流程所定义的语义耦合，增加了工作难度。

最终，由于该软件项目以这个庞大的 Listing 类为中心，我们撤销了重构，另辟蹊径解决了协调问题。当 Listing 发生变更时，持续集成服务器将自动发送一条消息到相关团队，以此鼓励积极的集成。因此，我们通过工程实践解决了协调问题，而不是通过调整架构。

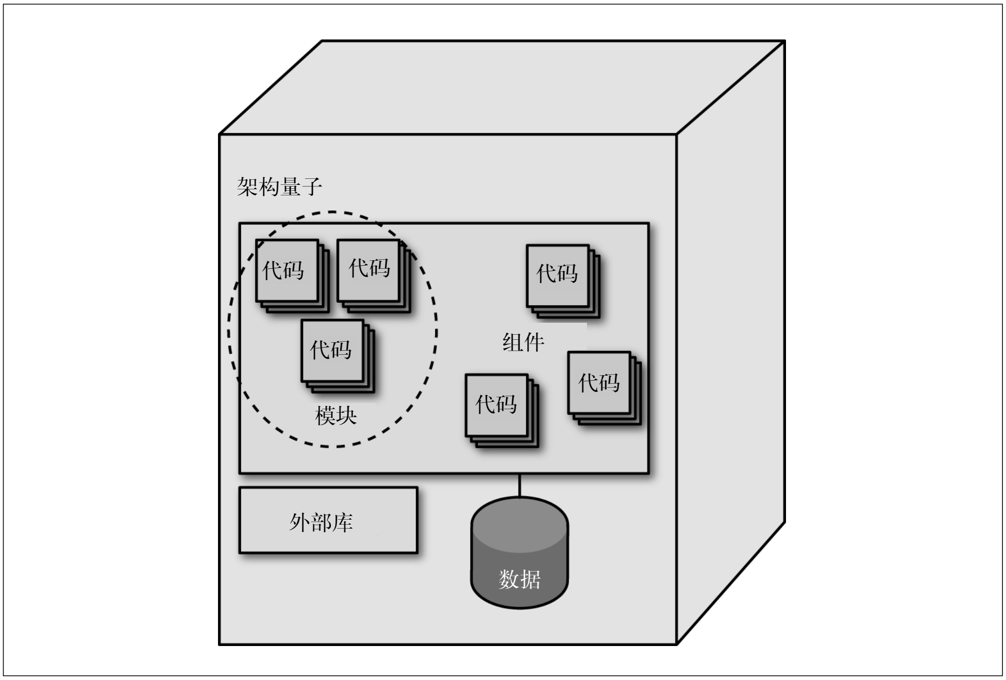


图 4-2：模块，组件和架构量子之间的关系

如图 4-2 所示，最外层的容器表示架构**量子**，即可部署单元，其包含系统正常运行所需的所有设施（包含数据）。架构量子内部存在一些**组件**，它们由代码（类、包、命名空间、函数等）构成。还有来自开源项目的外部组件**库**——可复用于给定平台打包好的组件。当然，开发人员可以随意组合这些基本构件。

4.3　不同类型架构的演进能力

软件架构之所以存在，部分原因是为了实现跨特定维度的某种演进——便于变更是架构模式的原因之一。架构模式不同，架构量子大小也不同，这影响着架构的演进能力。下面探究几个流行的架构模式，了解其架构量子的固有大小，同时基于 3 个演进条件（增量变更、适应度函数和适当的耦合）来介绍架构量子的大小对架构演进能力的影响。

需要注意的是，虽然架构模式对于成功演进至关重要，但是它并不是唯一的决定性因素。必须结合架构模式固有的特征和系统定义的附加特征才能完整定义演进性的各个维度。

4.3.1　大泥团架构

首先设想一种退化的情况，某个无法识别架构的混乱系统，俗称“大泥团反模式”。虽然其中还有一些典型的架构组件，例如框架和库，但并不是开发人员特意构建的。这些系统

高度耦合，当发生变更时会产生连锁副作用。开发人员创建了高度耦合且模块化很差的类。UI 和系统其他部分中隐藏着数据库模式，使它们难以变更。为了避免重构，数据库管理员花费了十年时间将紧密绑定的连接表拼接在一起。可能因为严格的预算限制，运维尽可能多地将系统挤在一起并以此来解决运维的耦合。

图 4-3 展示了某个大泥团架构中类的耦合情况，图中每个节点代表一个类，每条线（向内或向外）代表耦合，线的粗细程度表示连接的数量。

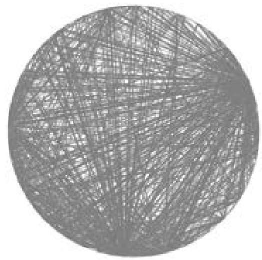


图 4-3：某个反常架构中的输入 / 输出耦合

改变图 4-3 所示的（来源于某个真实项目的）应用中的任意部分将面临严峻的挑战。类之间的耦合太多，导致我们无法在不影响其他部分的情况下修改系统的某一部分。因此站在演进能力的角度来看，这个架构表现极差。开发人员如果想修改整个应用的数据访问，必须找到所有存在的地方并修改它们，以免错过某些地方。

从演进的角度来看，这种架构在各个标准下都表现不佳。

- 增量变更

  对这种架构难以做任何变更。相关的代码散布于系统各个角落，这意味着修改其中一个组件将意外地破坏其他组件。修复这些破损会导致更多的破损发生，从而产生无尽的连锁反应。

- 通过适应度函数引导变更

  由于没有明确定义分区，我们很难为这种架构构建适应度函数。为了构建保护功能，开发人员必须确定需要保护的部分，但是在这种架构中，除了低级的函数或类之外不存在任何结构。

- 适当的耦合

  这种架构是**不当**耦合的典型。构建这样的软件没有任何架构优势。

在这样的糟糕状态下，变更困难且成本高。本质上，由于系统各部分间高度耦合，架构量子就是整个系统本身，没有哪个部分可以轻易改变，因为牵一发而动全身。

4.3.2　单体架构

单体架构的大量代码通常高度耦合。我们基于组织结构来介绍这种架构的几个变体。

1. 非结构化的单体架构

这种架构模式包含几种不同的变体，其中包括实质上由相互独立的类互相协调而构成的系统，如图 4-4 所示。

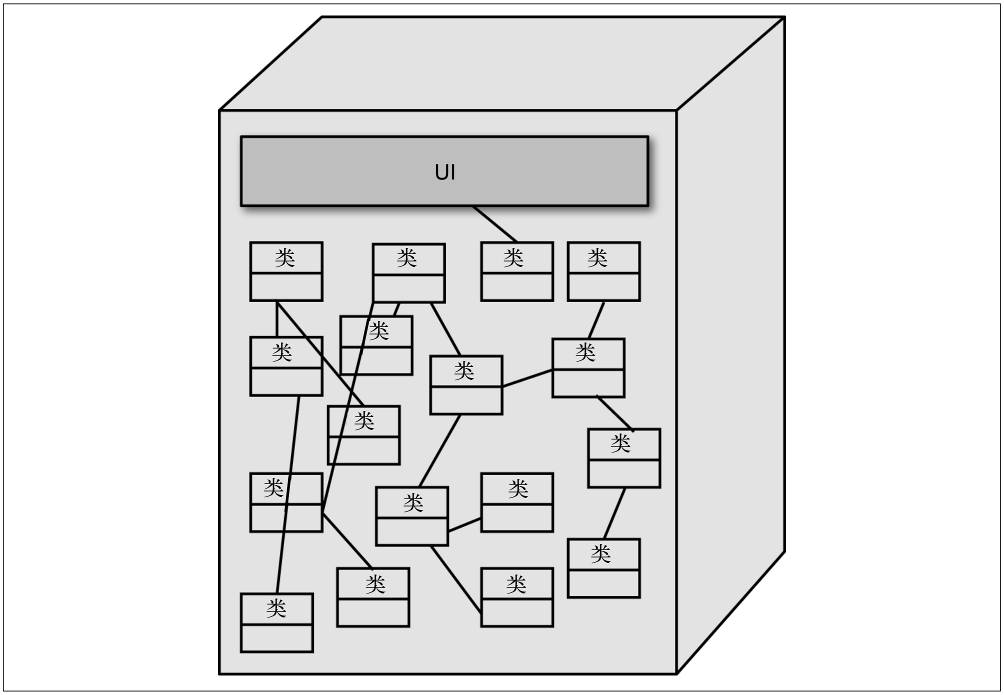


图 4-4：单体架构有时包含一系列关联松散的类

在图 4-4 中，不同的模块各自处理不同的任务，通过共用的类实现通用功能。在这种架构中，由于缺乏一致的总体结构而阻碍了变更。

- 增量变更

  巨大的架构量子阻碍了增量变更，因为高度耦合要求部署大块应用。组件之间存在高度耦合，这导致很难单独部署某个组件，因为需要变更其他组件。

- 通过适应度函数引导性变更

  为单体架构构建适应度函数很难，但并非不可能。因为这种架构模式存在了很长时间，可以用随之发展而来的很多工具和测试实践来构建适应度函数。然而，常见的引导性变更对象通常会成为单体架构的致命弱点，例如性能和伸缩性。虽然开发人员很容易理解单体架构，但难以构建良好的伸缩性和性能，这很大程度上源于它固有的耦合。

- ❑ 适当的耦合

  单体架构除了简单的类之外几乎没有内部结构，其耦合程度类似于大泥团架构。因此代码某处的变更可能对其中某个较远的部分产生意想不到的副作用。

尽管这种架构的演进能力略好于大泥团架构，但是这种架构很容易退化，因为几乎没有结构限制来防止其退化。

2. 分层架构

其他单体架构以更加结构化的方式创造出了分层架构，其中一个变体如图 4-5 所示。

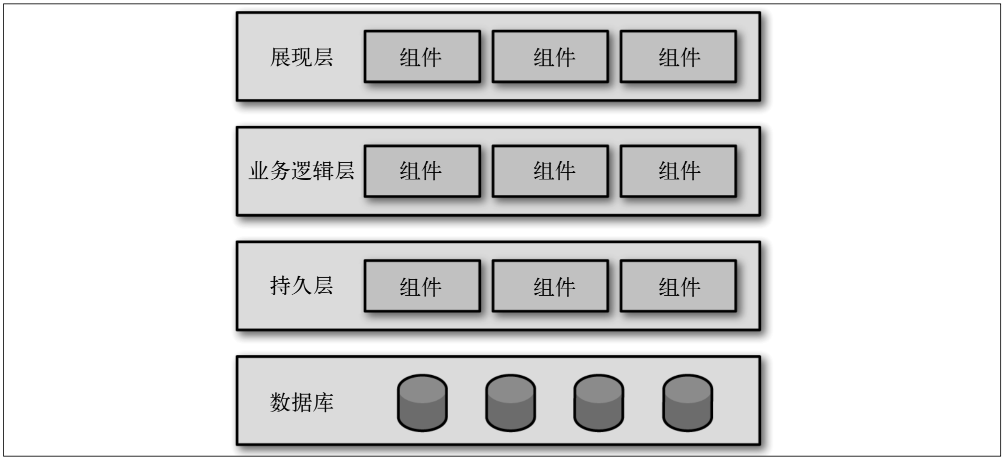


图 4-5：典型的分层单体架构

在图 4-5 中，每层代表一种技术能力，使得开发者能够轻易地置换技术架构功能。分层架构的主要设计准则是将不同的技术能力分隔到不同的层，每层职责各异。这种架构的主要优点是**关注点独立且分离**。每一层相对于其他层都是独立的，但能通过明确定义的接口互相访问。这使得对某一层的变更不会影响其他层，同时将相似的代码组织到一起，为该层的专业化和分离提供了空间。例如，持久层通常会封装所有数据保存的实现细节，使得其他层可以忽略这些细节。

无论哪种单体架构，架构量子本身就是应用，包括一些独立组件，例如数据库服务器。量子较大的系统是难以演进的。

- ❑ 增量变更

  开发人员发现变更这种架构很容易，特别是在将变更隔离到现有层的情况下。跨不同层的变更则会带来协调上的挑战，特别是在组织人员结构和架构分层类似的情况下（这反映了“康威定律”）。例如，某个团队能在不打扰其他团队的情况下替换整个持久层框架，因为他们可以在明确定义的接口背后完成这项工作。但是，当业务要求变更 `ShipToCustomer`（送货服务）时，该变更则会影响所有层，于是协调在所难免。

- ❑ 通过适应度函数引导性变更

  开发人员发现，在一个更加结构化的单体应用中编写适应度函数更为容易，因为这种架构的结构更明显。同时，将关注点分离到不同层使得开发人员能对更多部分进行隔离测试，便于构建适应度函数。

- ❑ 适当的耦合

  单体架构的一个优点是易于理解。了解设计模式等概念的开发人员能轻易将这些知识应用于分层架构中。这种易理解性很大程度上是因为开发者能轻松地访问所有代码。分层架构使得由层定义的技术架构划分更易于演进。例如，一个设计（并实现）良好的分层架构能让我们很容易地替换掉数据库、业务规则或其他任何层，并将副作用减至最小。

无论有意或无意，单体架构往往都高度耦合。当开发人员使用分层架构来分离关注点时（例如使用持久层去简化数据访问），该层通常会表现出内部高度耦合和外部低耦合。在层内，各组件为相同的目标合作，因此它们趋向于高度耦合。相反，开发人员通常会更仔细地定义各层之间的接口，在各层之间创建更低的耦合。

3. 模块化的单体架构

架构师们所赞赏的微服务的许多的优点也能在单体架构中实现，例如隔离性、独立性和小变更单元等，但前提是开发人员极其严格地处理耦合。需要注意的是，这个原则必须拓展到技术架构之外，囊括其他维度（特别是数据）。现代工具让代码易于复用，这使得开发人员很难在容易产生耦合的环境中实现适当的耦合。像示例 4-1 中的适应度函数，架构师在部署流水线中构建一道安全网，用以保持单体应用中组件间的依赖干净。

大部分现代编程语言都支持构建严格的可见性和连接规则。如果架构师和开发人员运用这些规则构建一个模块化的单体应用，那么构建出的架构会更具可塑性，如图 4-6 所展示的模块化良好的单体应用。

- ❑ 增量变更

  由于开发人员能够执行模块化，因此在此类架构中很容易进行增量变更。尽管在**逻辑上**功能被划分为不同的模块，但如果难以单独部署包含模块的组件，那么架构量子依然会很大。在模块化单体架构中，组件的可部署程度决定了增量变更的速度。

- ❑ 通过适应度函数进行引导性变更

  测试、度量及其他适应度函数在这种架构中更容易设计和执行，因为合理划分了组件，使得测试模拟和其他依赖于隔离层的测试技术更容易实现。

- ❑ 适当的耦合

  一个设计良好的模块化单体架构是适当耦合的好例子。每个组件在功能上是内聚的，组件之间的接口设计良好且耦合度低。

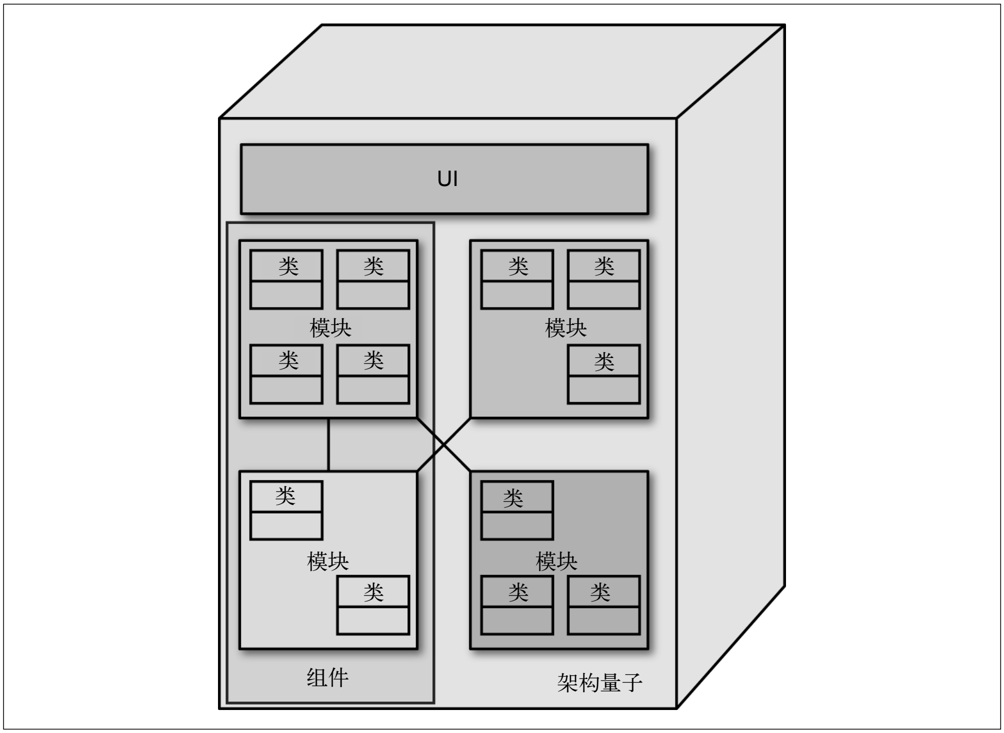


图 4-6：模块化的单体架构中包含功能的逻辑分组及模块间明确定义的边界

在开始一个新项目时，单体架构，特别是分层架构是普遍的选择，因为它的结构容易理解。但是由于性能下降、代码库过大和其他一系列因素，很多单体最终被取代而走到生命尽头。当前微服务架构是单体架构常见的迁移目标，但相比于单体架构，它在很多方面都更复杂，例如服务、数据粒度、运维、协调、事务等。如果开发团队难以构建最简单的架构，那么转向更复杂的架构又如何能解决问题呢？

> 如果无法构建单体应用，为什么你认为微服务能解决问题呢？
>
> ——Simon Brown

在重建昂贵的架构之前，提升现有架构的模块化程度能让架构师获益。如果已经没有可以提升的地方了，那么这便是开始重建更加复杂的架构的好时机。

4. 微内核架构

还有一种流行的单体架构——微内核架构，它通常出现在浏览器和集成开发环境（IDE）中，如图 4-7 所示。

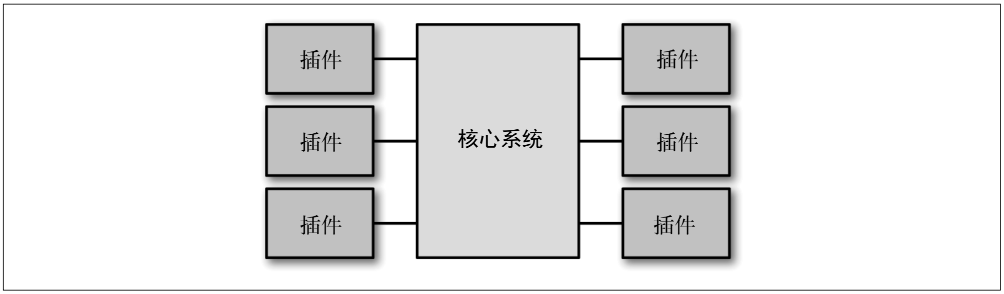


图 4-7：微内核架构

图 4-7 所示的微内核架构定义了一个核心系统，核心系统对外提供 API 来通过插件丰富其功能。在这种架构中架构量子大小有两种：一种来自核心系统，另一种来自插件。架构师通常将核心系统设计成单体应用，并在一些熟知的扩展点为插件创建钩子（hook）。我们通常把插件设计成独立且可单独部署的组件。因此，这种架构支持积极的增量变更，开发人员可以针对可测试性进行设计，更容易定义适应度函数。从技术耦合的角度来看，架构师往往将此类系统设计成低耦合，以保持插件相互独立，从而简化它们。

微内核架构的主要挑战围绕着契约，它是某种形式的语义耦合。为了发挥作用，插件必须和核心系统进行双向信息传递。只要插件不需要互相协调，那么开发人员就可以专注于插件与核心系统间的信息和版本控制。例如，大多数浏览器插件只和浏览器交互，而不和其他插件交互。

对于更复杂的微内核系统，插件间的通信无法避免，例如 Eclipse Java IDE。除了与文本文件交互之外，Eclipse 的核心不支持任何特定语言。所有复杂行为都是通过插件间的通信实现的。例如，在调试过程中编译器和调试器必须紧密协调。因为插件不应依赖其他插件工作，所以核心系统必须处理插件间的通信，这使得协调契约和版本控制等基本任务变得复杂。这个级别的隔离能让系统几乎无状态，虽然很理想，但通常不太可能实现。例如，在 Eclipse 中，插件运行通常依赖于其他插件，这导致围绕插件架构量子层面的传递依赖于管理。

通常，微内核架构包含一个注册表来跟踪安装的插件及其所支持的契约。在插件间建立明确的耦合加重了系统各部分间的语义耦合，进而导致架构量子变大。

微内核架构广泛应用于 IDE 工具，它也能应用于各种商业应用。例如，设想一家保险公司，其理赔的标准业务规则应用于整个公司，但是每个州可能有特别的法规。将该系统构建成微内核系统使得开发人员可以按需支持新的州，并在不影响其他州的情况下升级各州的行为，这得益于插件与生俱来的隔离性。

如果难以通过插件使技术架构演进，那么微内核架构是个不错的选择。由完全独立的插件组成的系统更易于演进，因为插件之间不存在耦合。但依赖彼此协作的插件会增加耦合，

进而阻碍系统演进。如果使用彼此交互的插件来设计系统，那么你还应该通过消费者驱动的契约模型构建适应度函数来保护那些集成点。微内核架构的核心系统通常很庞大，但是很稳定，因为大部分的变更应该发生在插件上（除非架构师将应用划分得很差）。因此，增量变更很简单：部署流水线触发对插件的变更并对其进行验证。

架构师通常不会在微内核技术架构中包含数据依赖，因此开发人员和数据库管理员必须单独考虑数据的演进能力。将每个插件视为限界上下文可以提高该架构的演进能力，因为这样可以降低内部耦合。例如，如果所有的插件将同一个数据库用作核心系统，那么开发人员必须注意插件在数据级别发生耦合的情况。如果各个插件完全独立，那么这样的数据耦合是不会发生的。

从架构演进的角度来看，微内核架构的理想特征如下所示。

- 增量变更

  一旦完成了核心系统，大多数行为应来自插件。如果插件都是独立的，那么增量变更会更容易。

- 通过适应度函数进行引导性变更

  通常在这种架构中构建适应度函数很简单，因为核心系统和插件是相对独立的。开发人员分别为核心系统和插件维护两套适应度函数。核心适应度函数守护核心系统的变更，包括伸缩性等部署问题。插件测试通常更简单，因为对领域行为的测试是隔离的。为了便于测试插件，开发人员需要很好地模拟核心系统。

- 适当的耦合

  微内核模式明确定义了这种架构的耦合特征。从耦合的角度来看，构建独立的插件使变更变得不重要。协调相互依赖的插件则更难。开发人员应该通过适应度函数来将相互依赖的组件正确地集成。

此类架构还应包含一些整体适应度函数来确保开发人员维持关键的架构特征。例如，单独的插件可能影响某个系统属性，比如伸缩性。因此，开发人员应该计划构建一套集成测试，把它作为整体适应度函数。当系统中存在相互依赖的插件时，开发人员还应该构建整体适应度函数来确保契约和消息的一致性。

4.3.3 事件驱动架构

事件驱动架构（EDA）通常通过消息队列将几个不相关的系统集成在一起。此类架构常用的实现方式有两种：代理模式和中介模式。两种模式的核心能力不同，因此我们分别讨论它们及其演进能力。

1. 代理模式

代理模式的事件驱动架构由如下架构组件构成。

- ❑ 消息队列

  消息队列由多种技术实现，例如 JMS（Java 消息服务）。

- ❑ 始发事件

  启动业务流程的事件。

- ❑ 流程内事件

  为了满足业务流程，在事件处理器之间传递的事件。

- ❑ 事件处理器

  事件处理器是活跃的架构组件，执行实际的业务流程。当两个处理器间需要协调时，它们通过队列传递消息。

图 4-8 展示了一个典型的代理模式的 EDA 工作流——保险公司的客户变更地址的过程。

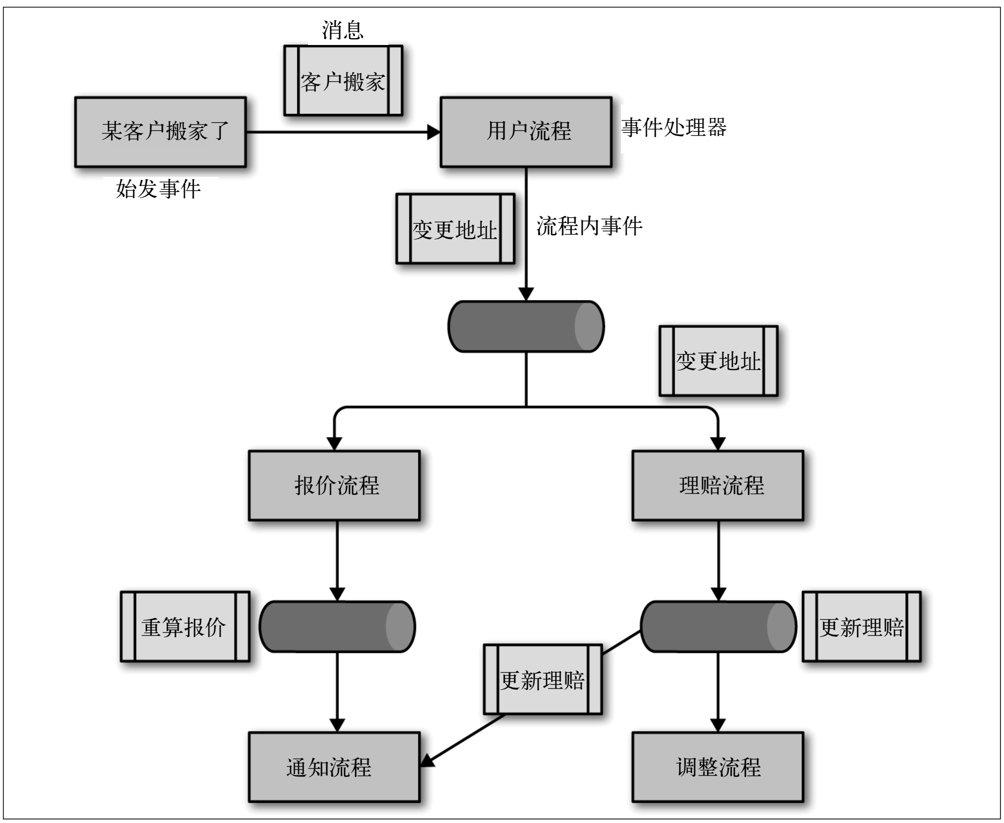


图 4-8：捕捉“客户搬家”事件的异步工作流

如图 4-8 所示，始发事件是客户搬家。该事件第一个的处理器是客户流程，它会更新内部地址记录。在完成更新后，它会向地址变更的消息队列中发布一条消息。报价流程和理赔流程将同时响应该事件，更新它们各自在意的信息。需要注意的是，由于服务间不需要协调，所以这些操作可以并行，这是这种架构的一个主要优点。一旦操作完成，每个处理器都会向相关队列中发布消息，例如通知。

代理模式的事件驱动架构在构建强大的异步系统时存在一些设计挑战。例如，由于缺少集中的中介，很难进行协调和错误处理。由于架构各部分高度分离，开发人员必须通过架构还原业务流程的功能内聚。因此，像事务这样的行为将更难实现。

尽管在实现上存在挑战，但它仍然是极具演进性的架构。开发人员可以通过向现有事件队列添加新的监听器来向系统添加新行为。例如，假设保险公司想审计所有索赔更新。开发人员可以在不影响现有工作流的情况下，向理赔事件队列添加一个审计监听器。

- 增量变更

  代理模式的事件驱动架构允许多种形式的增量变更。通常开发人员将服务设计为松散耦合的，便于独立部署。解耦转而使开发人员更容易做出无须中断的架构变更。为代理模式的事件驱动架构构建部署流水线是一项挑战，因为架构的本质是异步通信，但它很难测试。

- 通过适应度函数进行引导性变更

  对开发人员来说，在这种架构中编写原子适应度函数很容易，因为事件处理器的个体行为很简单。然而，在这种架构中编写整体适应度函数是必要且复杂的。整个系统的很多行为都依赖于松散服务之间的通信，这导致我们难以测试多层面的工作流。设想在图 4-8 所示的工作流中，开发人员可以轻易地通过事件处理器的单元测试来测试整个流程的各个独立部分，但测试所有流程更具挑战。有减轻此类架构中的测试挑战的一些方法。例如，关联 ID，即通过一个唯一的标识符标记每个请求，以跟踪跨服务的行为。类似地，合成事务让开发人员能在不进行真实交易的情况下测试协调逻辑，例如订购洗衣机。

- 适当的耦合

  代理模式的事件驱动架构所展现的低耦合增强了其进行演进式变更的能力。例如，想为这种架构添加新的行为，只需要将新的监听器添加到现有端点，这样不会影响现有的监听器。在这种架构中，服务和它们所维持的消息契约之间存在耦合，这是功能内聚的一种形式。适应度函数运用消费者驱动的契约等技术来帮助管理集成点，避免其被破坏。

在适合代理模式的 EDA 的业务流程中，事件处理器通常是无状态的、解耦的并且管理自身的数据。这使得演进更加容易，因为它的外部耦合更少，例如数据库耦合（第 5 章将讨论）。

2. 中介模式

另一个常见的 EDA 模式是**中介模式**，该模式包含一个额外的组件：作为中介的总线，如图 4-9 所示。

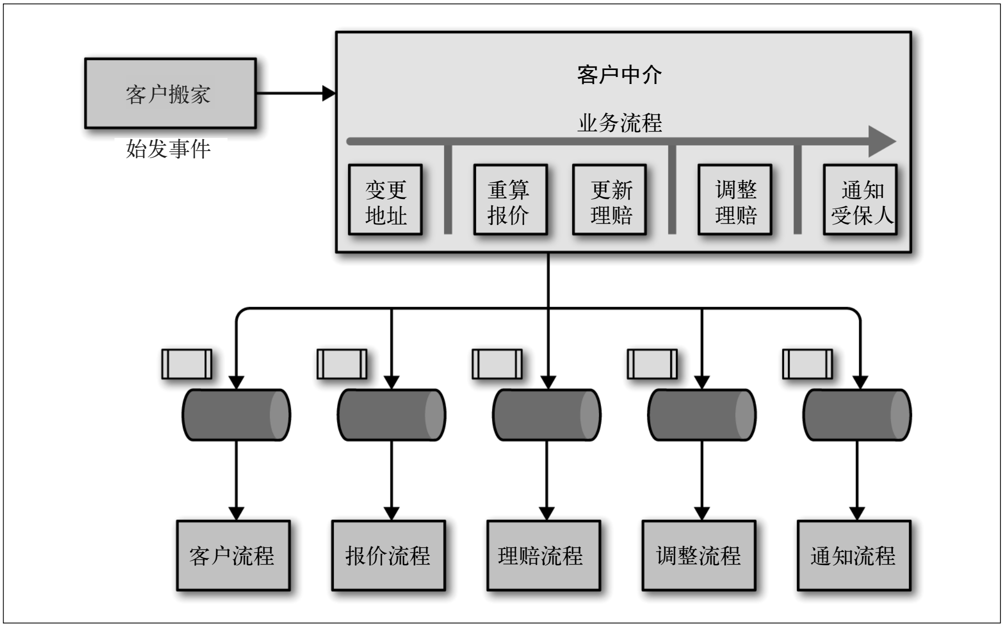


图 4-9：中介架构中的“客户搬家”流程

在图 4-9 中，中介响应始发的“客户搬家”事件并将工作流定义为：变更地址、重算报价、更新理赔、调整理赔和通知受保人。然后中介再将消息发布到队列中触发相应的事件处理器。虽然中介负责协调，但这仍是一种 EDA，能让大多数流程并行。例如，重算报价和调整理赔的流程。一旦完成所有任务，中介便会生成一条消息到通知受保人队列，让它生成一条单一状态消息。任何需要和其他处理器通信的流程都是经由中介处理的。通常在这种架构中处理器不会互相调用，因为中介定义了重要的工作流信息，从而避免了直接的通信。注意图 4-9 中中介区域的竖条，它表示并行的执行及服务请求和响应的协调。

事务性的协调是中介架构的主要优势。中介能保证流程的正确性，并生成一条单一状态消息发送给受保人。在代理事件驱动架构中，这样的协调更加困难。例如，要生成统一的通知消息，需要协调通知事件处理器或通过某个显式消息队列来处理这种聚合。虽然异步架构在协调和事务行为方面带来了挑战，但是它们的并行规模极佳。

- 增量变更

  和代理模式类似，在中介模式中，服务通常很小并且是独立的。因此，这种架构在进行增量变更时具有和代理版本相同的优势。

- 通过适应度函数进行引导性变更

  开发人员发现，相比代理模式，为中介模式构建适应度函数更容易。两种模式在测试单个事件处理器上大致相同。然而，构建全系统适应度函数会更容易，因为开发人员可以依赖中介进行协调。例如，在保险工作流中，开发人员可以编写测试并能轻易知晓整个过程是否成功，因为中介在协调这一切。

- 适当的耦合

  虽然中介便利了很多测试场景，但也增加了耦合，妨碍了演进。中介包含了重要的领域逻辑，使得架构量子增大，导致了各个服务间的相互耦合。在这种架构中，当有开发人员进行变更时，其他开发人员必须考虑变更对工作流中其他服务产生的副作用、增加的耦合。

从演进的角度来看，由于降低了耦合，代理架构具有明显优势。在中介模式中，中介充当了耦合点，它将所有受影响的服务绑定在一起。在代理模式中，行为可以通过向已有消息队列添加新的处理器来演进，而不影响其他消息队列。(除了通过流量使队列的负载过重的情况，这种情况可以通过多种架构模式或适应度函数来解决)。由于代理模式在本质上是解耦的，因此更易于演进。

这是权衡架构的典型例子。代理模式在演进能力、异步性、伸缩性及其他一些所期望的特征上具有优势，但不擅长协调事务等一些基本任务。

4.3.4 服务导向架构

现有的服务导向架构（SOA）有很多种类，包括一些混合架构。下面介绍一些常见的架构模式。

1. 企业服务总线驱动的 SOA

有种构建 SOA 的特殊方式在几年前流行起来，那便是通过**服务总线**构建围绕服务和协调的架构，通常称为**企业服务总线**（ESB）。服务总线充当复杂事件交互的中介，并处理其他典型的集成架构中的各种琐事，例如消息转换、编排等。

虽然企业服务总线架构通常使用和事件驱动架构相同的构件，但服务的组织方式不同。企业服务总线架构基于严格定义的服务分类方法来组织服务。其样式因组织而异，但都是根据复用性、共享概念及服务范围划分服务。图 4-10 展示了一个典型的 ESB 驱动的 SOA。

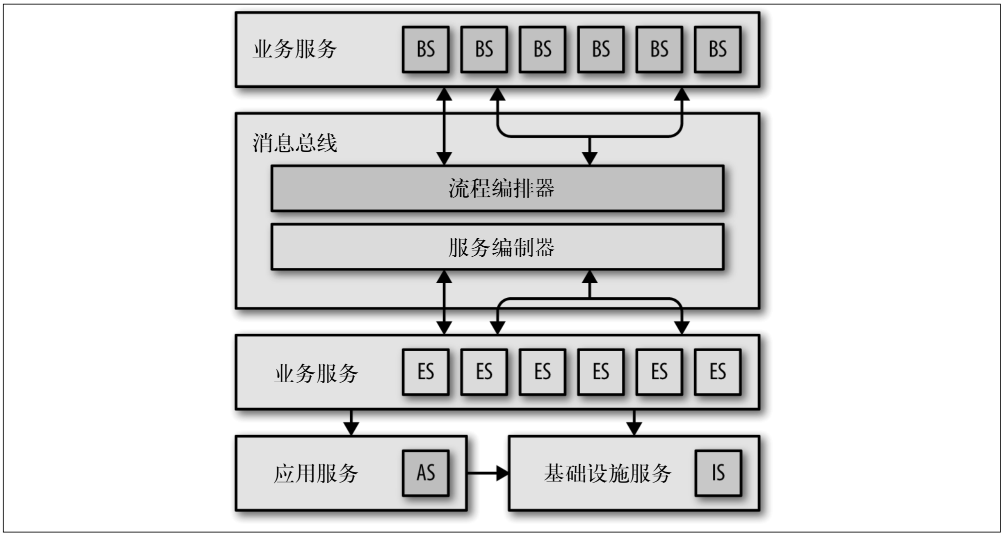


图 4-10：ESB 驱动的 SOA 的典型服务分类方法

在图 4-10 中，架构的每一层都有特殊的职责。业务服务抽象定义了业务的粗粒度功能，有时业务人员使用标准工具来定义业务服务，例如业务过程执行语言（BPEL）。业务服务的目标以抽象的方式捕捉公司所做的事情。想确定这些服务是否在正确的抽象级别定义，将每个服务名套入“我们的业务是不是……？”（例如创建报价或执行交易）问题中向自己提问。你会发现，虽然开发人员可能为了创建报价而调用创建用户（`CreatCustomer`）服务，但那只是一个必要的中间步骤，并不是主要的业务推动力。

抽象的业务服务必须调用代码来执行行为，这便是**企业服务**。它由不同的服务团队负责，旨在实现共享。这些团队的目标是构建可复用的服务集成架构，架构师可以通过编排将服务“缝合”在一起形成业务实现。开发人员以高度复用为目标并设计相应的企业服务。（要理解其经常失败的原因，参见 7.14 节。）

有些服务并不需要高度可用。例如，系统的某个部分可能需要地理位置，但是没有重要到需要投入资源将其构建成完整的企业服务的程度。图 4-10 左下角的**应用服务**便是处理这类情况。这些服务由特定的应用团队负责，它们与某个特定的应用上下文绑定，并不会被复用。

基础设施服务是共享的服务，由基础设施团队负责。它处理一些非功能需求，例如监控、日志、认证 / 授权等。

ESB 驱动的 SOA 的标志是消息总线，它负责以下各类任务。

- 中介和路由

  消息总线能够定位服务并与服务通信。通常，消息总线会维护一张注册表，涵盖服务的物理地址、协议以及调用服务所需的其他信息。

- 流程编排与编制

  消息总线将企业服务组合到一起并管理任务，例如服务调用顺序。

- 消息增强和转换

  集成总线的优势之一是能够代表应用处理通信协议及其他信息的转换。例如，支持 HTTP 协议的服务 A（ServiceA）想调用仅支持 RMI/IIOP 协议的服务 B（ServiceB）。当需要此类转换时，开发人员可以配置消息总线，在无形之中完成此类消息转换。

ESB 驱动的 SOA 的架构量子很大。它基本包含了整个系统，和单体应用差不多，但由于它是分布式架构，因此更为复杂。在 ESB 驱动的 SOA 中进行演进式变更非常困难，因为在促进服务复用的同时，其服务分类方法会阻碍普通的变更。例如，在某个 SOA 中，商品结账（CatalogCheckout）这一领域概念被打散到了整个技术架构中。仅变更商品结账就需要架构各部分进行协调，这些部分往往由不同的团队负责，从而会产生协调摩擦。

比较商品结账的这种表达和微服务中限界上下文的划分。在微服务架构中，每个限界上下文表示一个业务流程或工作流。因此，开发人员会围绕商品结账构建限界上下文。商品结账可能需要客户（Customer）详情，但是每个限界上下文有自己的实体。如果其他限界上下文同样拥有客户的概念，开发人员不会将它们统一成共享的 Customer 类。恰恰相反，这正是 ESB 驱动的 SOA 的首选方式。如果商品结账和送货订单（ShipOrder）限界上下文需要共享客户信息，微服务会传递消息而不是尝试将它们统一成单一的表达。

ESB 驱动的 SOA 没有展现出任何演进式架构的特性，所以它在演进性的各个方面得分都不高就不足为奇了。

- 增量变更

  虽然对复用和隔离资源有完善的技术服务分类方法，但这种架构却严重地阻碍了对业务领域进行最常规的变更。大多数 SOA 团队都是按照架构划分的，这导致进行常规的变更需要大量的协调工作。而且 ESB 驱动的 SOA 也是难以操作的。通常它由多个物理部署单元组成，这给协调和自动化带来了挑战。没人会为了敏捷性和操作的易用性而采用企业服务总线。

- 通过适应度函数进行引导性变更

  在 ESB 驱动的 SOA 中，通常很难进行测试。各部分都不完整，都是某个巨大工作流的一个环节，通常无法单独测试它们。例如，某个为了复用而设计的企业服务，测试其核心行为通常很困难，因为它可能只是各个工作流的一部分。为其构建原子适应度函数几乎不可能，这导致需要大规模的整体适应度函数进行端到端测试来完成大部分验证工作。

- 适当的耦合

  从潜在的企业级复用来看，这种奢侈的分类方法是合理的。如果开发人员可以准确地提炼出每个工作流中可复用的精华，那么最终他们就能够一劳永逸地构建出企业的所有行

为，而将来的应用开发就变成了连接现有服务。构建 ESB 驱动的 SOA 的目标不是为了系统各部分能够独立演进，所以在这方面它表现得非常糟糕。针对分类复用的设计，损害了它在架构级别进行演进变更的能力。

软件架构并不是在真空环境中构建的，它们始终反映其所处的环境。例如，当 SOA 还是流行的架构样式时，企业不会使用开源的操作系统，所有基础设施都需要付费获得使用许可，且非常昂贵。十年前，如果开发人员提议使用微服务、在具有单独操作系统和主机的实例上运行服务，那么他会被运营中心嘲笑，因为这种架构在当时贵得离谱。由于软件开发环境的动态平衡，新的环境孕育新的架构。

虽然可能由于某些原因架构师选择了 ESB 驱动的 SOA，例如处理集成繁重的环境、规模、服务分类或其他合理的原因，但决不是为了演进能力，因为 ESB 驱动的 SOA 并不适合演进。

2. 微服务架构

将持续交付的工程实践和限界上下文的逻辑划分相结合，便形成了微服务的思想基础以及架构量子概念。

在分层架构中，关注点在**技术**层面，或者说在应用各部分的工作方式，例如持久性、UI、业务规则等。大部分软件架构都关注这些技术维度。然而，还有另一种视角。假设应用中的一个关键限界上下文是“结账”，在分层架构中它应该身处何处呢？与“结账”类似的领域概念在分层架构的各层都会涉及。由于架构被技术分层隔开，所以在这种架构中没有清晰的**领域**维度的概念，如图 4-11 所示。

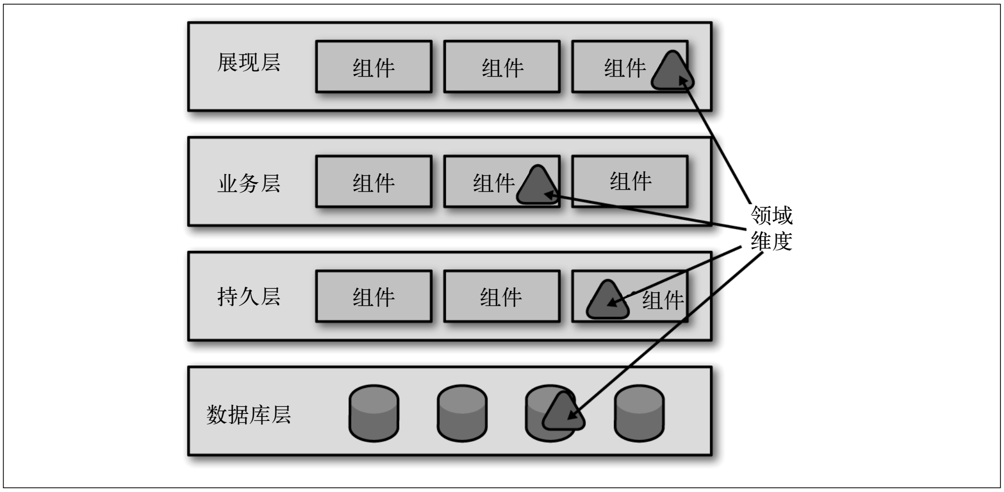


图 4-11：领域维度嵌入到了技术架构中

在图 4-11 中，“结账”这个领域概念的展现逻辑、业务规则和持久性分别由架构的不同层处理。由于分层架构的设计初衷不是容纳领域概念，因此开发人员为了变更领域必须修改每一层。从领域的视角来看，分层架构不具有演进性。在高度耦合的架构中，由于各部分之间的高耦合度，开发人员很难对其进行变更。然而，在大部分项目中，通常会围绕领域概念进行变更。 如果软件开发团队的组织形式像分层架构那样相互独立，那么变更“结账”这个领域概念需要跨团队进行协调。

相反，设想一个主要通过**领域**维度进行划分的架构，如图 4-12 所示。

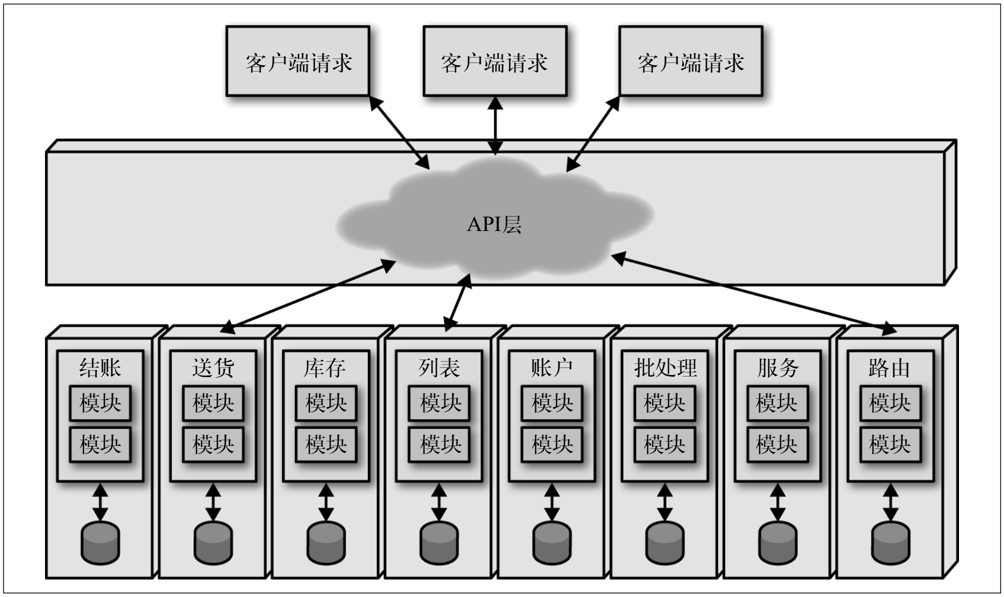


图 4-12：微服务架构根据领域划分，技术架构嵌于各个服务中

如图 4-12 所示，每个服务都围绕 DDD 的领域概念定义，并将技术架构和所依赖的其他组件（例如数据库）封装到限界上下文中，构建了高度解耦的架构。每个服务包含其限界上下文的所有部分，并通过消息（例如 REST 或消息队列）和其他限界上下文进行通信。因此，服务不需要知道另一个服务的实现细节（例如数据库模式），从而避免了不当的耦合。该架构的运作目标是用一个服务取代另一个服务而不影响其他服务。

微服务架构通常遵循以下七个原则。

- 围绕业务领域建模

  微服务设计的重点是基于业务领域，而不是基于技术架构。因此，架构量子反映了限界上下文。一些开发人员错误地认为限界上下文代表某个单独的实体，例如客户。相反，它代表某个业务上下文或工作流，例如商品结账。微服务的目标是创建有用的限界上下文，而不是让开发人员构建更小的服务。

- 隐藏实现细节

  微服务的技术架构封装在基于业务领域的服务边界中。每个领域形成一个物理限界上下文。服务间通过传递消息或资源来集成，而不是通过暴露实现细节集成，例如数据库模式。

- 自动化文化

  微服务架构支持持续交付，它使用部署流水线严格地测试代码，并将一些任务自动化，例如服务器准备和部署。在高速变化的环境中，自动化测试能发挥巨大作用。

- 高度去中心化

  微服务形成了一种**无共享**架构，其目标是尽可能地减少耦合。通常重复好于耦合。例如，商品结账和送货服务都包含项（`Item`）的概念。由于在两个服务中，这个概念的名称和属性相同，因此，为了节省时间和工作量，开发人员想在两个服务中复用这个概念。但这样反而会增加工作量，因为变更会影响所有共用该组件的团队。不仅如此，在变更某个服务时，开发人员还要担心对共享组件的修改。相反，如果每个服务有其自己的项，并将其所需的信息从商品结账传递到送货服务，而不需要耦合组件，那么它们便可以各自进行变更。

- 独立部署

  开发人员和运维人员希望可以独立部署每个服务（包括基础设施），反映了服务间的物理限界上下文。微服务架构的一个明显的优点是开发人员可以在不影响其他服务的情况下部署某个服务。而且，开发人员通常会自动化所有的部署和运维任务，例如并行的测试和持续交付。

- 隔离失败

  开发人员会在微服务上下文中和服务间的协调中隔离失败。每个服务都应该处理合理的错误场景并在可能的情况下将其恢复。很多 DevOps 的最佳实践通常在这种架构中出现，例如熔断器模式、舱壁模式等。很多微服务架构遵循着响应式宣言（reactive manifesto），它是一系列运作和协调原则，遵循这些原则可以构建出更加强大的系统。

- 高度可观察

  开发人员不能期望人工监控成百上千个服务（一个开发人员无法观察多个 SSH 终端会话）。因此，在微服务架构中监控和日志成了首要问题。如果运维人员无法监控某个服务，那么它相当于不存在了。

微服务的主要目标是通过物理限界上下文来隔离领域及理解问题领域。因此，它的架构量子就是服务，这使得它成为了演进式架构的优秀示例。如果某个服务为了演进而改变其数据库，这不会影响其他服务，因为其他服务不需要知道该服务的实现细节，例如数据库模式。当然，负责变更的开发人员必须向服务间的集成点发送相同的信息（希望**适应度函数保护它**，例如**消费者驱动的契约**），使得调用方的开发人员完全不用了解发生的变更。

鉴于我们将微服务看作演进式架构的典范，从演进的角度来看，它的优异表现不足为奇。

- 增量变更

  在微服务架构中，从各方面来看进行增量变更都很容易。每个服务围绕领域概念形成了限界上下文，使得变更只会影响服务所处的上下文。微服务架构强烈依赖于**持续交付**的自动化实践，利用部署流水线和现代 DevOps 实践。

- 通过适应度函数进行引导性变更

  开发人员可以很容易地为微服务架构构建原子适应度函数和整体适应度函数。每个服务有着明确定义的边界，开发人员可以在服务组件内进行各种级别的测试。服务间需要通过集成来相互协作，这些集成点也需要测试。幸运的是，随着微服务的发展，先进的测试技术也不断涌现。

- 适当的耦合

  微服务的耦合通常有两种：**集成**和**服务模板**。很明显，集成耦合的服务间需要互相调用来传递信息。另一种耦合，服务模板，用于防止有害的重复。如果各种设施能够在微服务内部管理并保持一致，开发人员和运维人员就能从中受益。例如，每个服务都需要包含监控、日志、认证 / 授权和其他一些基本能力。如果将它们交给各个服务团队负责，那么保证兼容性及生命周期管理（例如升级）将会非常困难。通过在服务模板中定义适当的技术架构耦合点，并让基础设施团队管理这些耦合，就能使各个服务团队免于这些苦恼。领域团队只需要扩展这些模板并编写自己的业务行为。基础设施升级后，在下一次部署流水线执行时服务模板将自动采用新行为。

微服务架构中的物理限界上下文正好与架构量子概念吻合，架构量子是一种具有高功能内聚性并在物理上解耦的可部署组件。

严格按照领域限界上下文进行服务划分是微服务架构的一个关键原则。微服务将技术架构内嵌到领域中，遵循 DDD 的限界上下文原则，对各个服务进行物理隔离，使得微服务在技术上成为**无共享**架构。每个服务在物理上都是分离的，可以轻松地替换和演进。由于每个微服务都在其限界上下文中内嵌了技术架构，它们都能以必要的方式演进。因此，微服务演进性的维度与其服务的数量相当，开发人员可以单独处理每个服务，因为每个服务都是高度解耦的。

“无共享”和适当的耦合

架构师通常将微服务称为“无共享”架构。这种架构的主要优势是在技术架构层面完全解耦。但是对耦合不满的人通常会提到“不当的耦合”。毕竟，一个没有耦合的软件系统也强不到哪里去。这里的“无共享”实际上是指“没有混乱的耦合点”。即便在微服务中，一些事物仍然需要共享和协调，比如工具、框架、库等。举例来说，日志、

监控、服务发现等。如果某个服务团队忘记为他们的服务添加监控能力，那么在部署时会很麻烦。在微服务架构中，如果无法监控某个服务，那么该服务将消失不见。

在微服务中，服务模板是解决这类问题的常见方案，例如 DropWizard 和 Spring Boot。DevOps 团队能够利用这些框架在服务模板中构建一致的工具、框架、版本等。服务开发团队使用这些模板承载他们的业务行为。当监控工具需要升级时，服务团队可以在服务模板内协调更新，而无须打扰其他团队。

既然微服务架构的优点如此明显，为什么之前开发人员没有采用它呢？十年前，自动化环境准备是行不通的。操作系统需要付费购买使用许可，基本上无法自动化。现实的限制因素（例如预算）影响着架构，这也是开发人员构建精细的、在技术层面隔离的共享资源架构的原因之一。如果运维成本高且繁琐，那么架构师就会构建出像基于 EBS 的 SOA 这样的架构。

持续交付和 DevOps 的发展为软件开发的动态平衡增添了新的因素。如今，我们可以将主机的定义用于版本控制并将自动化运用到极致。部署流水线并行启动多个测试环境来支持安全的持续部署。由于大部分的软件栈都是开源的，所以软件使用许可等问题不再影响架构。社区对软件开发领域出现的新能力做出反应，产生了更加以领域为中心的架构样式。

在微服务架构中，领域中封装了技术架构和其他架构，使得跨领域维度的演进更容易。关于架构，没有所谓“正确”的观点，它只是反映了开发人员构建项目的目标。如果仅关注技术架构，那么跨该维度的变更将会更容易。然而，一旦忽略了领域视角，那么跨领域维度的演进将和大泥团架构差不多。

系统的各部分如何在无意间互相耦合，这是在架构层面影响应用程序演进能力的一个主要因素。例如，在分层架构中，架构师有意地将某些层耦合在一起。然而，无意地将领域维度耦合到了一起，导致在领域维度难以演进。这是由于架构师围绕技术架构分层来设计，而不是围绕领域。因此，演进式架构的一个重要方面就是跨维度的**适当耦合**。第 8 章将讨论如何在实际应用中确定和利用架构量子边界。

3. 基于服务的架构

另一种常用于迁移的架构是**基于服务的架构**，它和微服务相似，但有三个明显的区别，分别是服务粒度、数据库范围和集成中间件。基于服务的架构同样以领域为中心，但当开发人员将现有应用重建为更具演进性的架构时，它能解决开发人员所面临的一些挑战。

- 更大的服务粒度

  这种架构中的服务往往更大，相较于纯粹围绕领域概念的服务，其服务粒度更像一个“单体应用”。虽然它仍以领域为中心，但是更大的服务导致变更单元（开发、部署、耦合以及其他一系列因素）也更大，增加了变更的难度。当架构师评估一个单体应用时，他们通常会观察围绕常见领域概念的粗粒度进行划分，例如结账或送货，这样便能很好

地完成架构的首次划分。基于服务的架构在运维隔离上和微服务的目标相同，但是更难实现。由于服务变大，开发人员必须考虑更多的耦合点，并且更大的代码段本身就更为复杂。理想情况下，架构应该支持和微服务一样的部署流水线和小的变更单元。当开发人员修改某个服务时，它应该触发部署流水线来重建相关服务，包括应用。

- 数据库作用域

  无论服务的分解程度如何，基于服务的架构往往都使用单体数据库。在很多应用中，将多年（甚至十多年）难以管理的数据库模式重建为原子大小的微服务是不可行甚至不可能的。虽然在某些情况下无法分解数据可能造成不便，但在某些问题域中是不可能的。事务完整性很强的系统不太适合微服务，因为在服务间进行事务行为协调的成本太高了。由于对数据库的要求更宽松，有着复杂事务需求的系统更适合基于服务的架构。

虽然数据库保持单一整体，但依赖于数据库的组件可能变得更为细化。因此，尽管服务和基础数据之间的映射可能改变，但这所需的重建较少。第 5 章将介绍演进式数据库设计。

- 集成中间件

  微服务和基于服务的架构之间的第三个区别涉及通过中介（如服务总线）来进行外部协调。开发人员在构建全新的微服务应用时不用担心老的集成点，然而很多环境中仍然有大量遗留系统在做着有价值的工作。集成总线（如企业服务总线）擅长将不同协议和消息格式的各种服务整合到一起。如果架构师发现开发环境中集成架构的优先级最高，可以使用集成总线添加和变更相关服务。

使用集成总线是一个典型的架构折中。使用集成总线，开发人员能够以更少的代码来将应用集成到一起，并能使用总线来模仿服务间的事务协调。然而使用总线增加了组件间的架构耦合，在不与其他团队协调的情况下，开发人员无法独立完成变更。适应度函数可以降低一些协调成本，但是开发者带来的耦合越高，系统演进就越难。

下面是我们对基于服务的架构的演进性评估的结果。

- 增量变更

  在这种架构中进行增量变更相对可行，因为每个服务都是以领域为中心的。软件项目中的大多数变更都是围绕领域发生的，在变更单元和部署量子之间保持了一致性。由于服务通常会更大，所以无法像微服务那样敏捷，但还是保留了微服务的很多优点。

- 通过适应度函数引导变更

  开发人员发现，在基于服务的架构中构建适应度函数通常比在微服务中构建更难，这是由于更高的耦合（通常是数据库耦合）和更大的限界上下文。高耦合的代码通常使编写测试更困难，同时数据耦合度的上升也会导致一系列问题。基于服务的架构中创建的限界上下文越大，系统内部的耦合点就越多，使得测试和其他一些诊断变得更复杂。

- 适当的耦合

  耦合通常是开发人员选择基于服务的架构而不是微服务架构的原因，例如，分解数据库模式的难度太大、重建单体应用时面临的高度耦合问题等。构建以领域为中心的服务有助于确保适当的耦合，同时服务模板有助于构建适当的技术架构耦合。

基于服务的架构内在的演进能力肯定比 ESB 驱动的 SOA 架构要好。开发人员偏离限界上下文的程度决定了架构量子的大小和破坏性耦合的数量。

基于服务的架构在纯粹的微服务思想和很多项目的实现之间做出了很好的折中。通过放宽对服务大小、数据库独立性和偶然但有用的耦合的限制，该架构解决了微服务中最另人头疼的问题，同时还保留了许多长处。

4.3.5 “无服务”架构

“无服务”架构是软件开发动态平衡中最近出现的变化，它的两大含义都适用于演进式架构。

BaaS（后端即服务）是那些明显或从根本上依赖于第三方应用或云端服务的应用。其简单示例如图 4-13 所示。

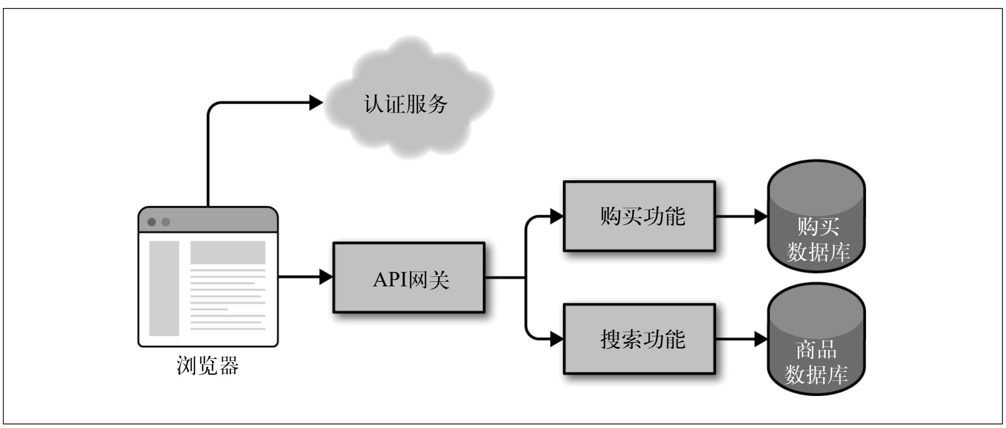


图 4-13：无服务 BaaS

如图 4-13 所示，开发人员编写少量代码甚至无须编写代码。架构由相连的服务组成，包括认证、数据传输和其他集成架构组件。这种架构之所以吸引人是因为组织编写的代码越少，他们需要维护的代码就越少。然而，重度集成的架构有其自身的挑战。

另一类无服务架构是 FaaS（功能即服务），它对基础设施要求不高（至少是在开发人员看来），为每个请求提供基础设施，自动处理水平扩展，还承担环境准备和其他一系列管理职责。在 FaaS 中，功能由服务提供商所定义的事件类型触发。例如，AWS（亚马逊云服

务）是知名的 FaaS 提供商，它提供了不同类型的事件源，如文件更新（在 S3 上）、时间（计划任务）和新添加到消息总线的消息（如 Kinesis）等。提供商通常会对功能进行一些限制，例如对请求处理的时长和其他围绕状态的限制。因此，通常假设 FaaS 功能是无状态的，因而调用者需要处理状态。

- 增量变更

  在无服务架构中，增量变更需要重新部署代码，基础设施相关的所有问题都在“无服务”的抽象背后。这种架构非常适用于部署流水线，在开发人员开展变更时使用部署流水线进行测试和增量的部署。

- 通过适应度函数引导变更

  在此类架构中，为了使集成点保持一致，适应度函数至关重要。因为服务间的协调是关键，开发人员需要编写更高比例的整体适应度函数，并在多个集成点上下文中运行它们，以保第三方 API 不变。架构师频繁地在集成点之间构建防腐层，以避免“供应商为王”反模式（7.11 节将讨论）。

- 适当的耦合

  从演进式架构的角度来看，FaaS 吸引人的原因是它消除了考虑因素中多个不同的维度，例如技术架构、运维和安全问题等。虽然这种架构可能易于演进，但在实际考虑方面受到了严重的限制，并将大量复杂问题转嫁给了调用方。例如，虽然 FaaS 能处理弹性的水平扩展，但是调用方必须处理所有的事务行为和其他复杂的协调工作。在传统应用中，通常由后端处理事务协调。然而，如果 BaaS 不支持该行为，那么协调工作就必然会转嫁给用户接口（服务请求者）。

架构师选择架构前，必须针对需要解决的现实问题评估架构。

> 确保架构与问题领域匹配。不要尝试强行使用不合适的架构。

虽然无服务架构有很多吸引人的特性，但也存在限制。尤其是像这种无所不包的解决方案通常会遇到 7.1.3 节将讨论的问题。团队需要构建的大部分内容都很快捷，但有时构建完整的方案会令人很苦恼。

4.4 控制架构量子大小

架构量子的大小很大程度上决定了开发人员进行演进式变更的难易程度。大的架构量子很难演进，例如单体架构和 ESB 驱动的 SOA 架构，因为每次变更都需要进行协调。低耦合的架构为轻松的演进提供了更多途径，例如代理模式的 EDA 和微服务。

架构演进的结构限制取决于开发人员处理耦合和功能内聚的水平。如果开发人员构建出的模块化组件系统具有明确定义的集成点，那么演进会更容易。例如，如果开发人员构建了一个单体应用，但是致力于良好的模块化和组件分离，那么该架构将更易于演进，因为解耦使得它的架构量子更小。



架构量子越小，架构的演进能力越强。

4.5 案例分析：防止组件循环依赖

PenultimateWidgets 正在开发几个单体应用。在设计组件时，架构师的目标是构建独立的组件，因为代码越分离，变更越容易。依托强大 IDE 的语言常会遇到一个的问题——**包循环依赖**，如图 4-14 所示。

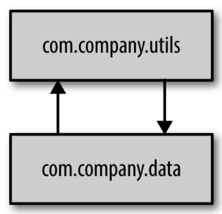


图 4-14：包循环依赖

在图 4-14 中，包 `com.company.data` 从 `com.company.utils` 包中导入依赖，同时 `com.company.utils` 包又从 `com.company.data` 中导入依赖，两个组件在对方缺失的情况下都无法工作，从而导致了循环的组件依赖。显然，循环降低了可变性，因为一个复杂的循环网络会增加增量变更的难度。像 Java 和 C# 语言，它们的开发环境会（通过内置于 IDE 中的代码洞察助手）建议缺少的导入来帮助开发人员。在 IDE 的帮助下，开发人员在日常编码过程中隐式地导入了很多依赖，使得包循环依赖很难避免，因为开发工具在暗中较劲。

在这些系统上工作的 PenultimateWidgets 架构师们担心开发人员会意外引入组件循环依赖的问题。幸运的是，有一种机制，即适应度函数，有助于防范那些损害系统演进能力的因素。与其因为 IDE 产生了某些副作用而放弃它所带来的好处，不如通过适应度函数构建一张工程安全网。一些商业和开源工具能帮助很多流行的平台解开依赖循环。其中一些是以静态代码分析工具的形式查找依赖循环，其他的则提供待重构列表来帮助开发人员解决这些问题。

在移除循环依赖后，如何防止开发人员习惯性地引入一个新的循环依赖呢？编码规范并不

能很好地解决这类问题，因为开发人员很难在编码过程中记得这些繁杂的守则。相反，他们往往通过构建测试和其他验证机制来防范这些过于“热心”的开发工具。

PenultimateWidgets 的开发人员使用了 JDepend，它是 Java 平台上一个流行的开源工具，包含了文字和图形界面来帮助分析依赖。由于 JDepend 是用 Java 编写的，开发人员可以利用其 API 来编写结构测试。参见示例 4-1 中的测试案例。

示例 4-1　使用 JDepend 通过编程的方式识别循环依赖

```
import java.io.*;
import java.util.*;
import junit.framework.*;

public class CycleTest extends TestCase {
    private JDepend jdepend;

    protected void setUp() throws IOException {
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/util/classes");
        jdepend.addDirectory("/path/to/project/web/classes");
        jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    /**
     测试不含任何包依赖循环的包。
     */
    public void testOnePackage() {
        jdepend.analyze();
        JavaPackage p = jdepend.getPackage("com.xyz.thirdpartyjars");
        assertEquals("Cycle exists: " + p.getName(),
                false, p.containsCycle());
    }

    /**
     测试不是为分析包而设置的包依赖循环。
     */
    public void testAllPackages() {
        Collection packages = jdepend.analyze();
        assertEquals("Cycles exist",
                     false, jdepend.containsCycles());
    }
}
```

在示例 4-1 中，开发人员将包目录添加到 JDepend。接着，正如单元测试 `testAllPackages()` 所展示的那样，开发人员可以测试单个包或整个代码库。一旦项目完成了确定并移除循环依赖的艰巨任务，便可以将单元测试 `testAllPackages()` 作为系统架构适应度函数，以防止循环依赖再次发生。

第5章

演进式数据

Pramod Sasalagec 参与编写

在现代软件项目中，关系型和其他类型的数据存储无处不在，这种形式的耦合通常比架构耦合更成问题。在构建可演进的架构时，需要考虑**数据这**一重要维度。本书不会涉及演进式数据库设计的各个方面，只介绍影响演进式架构的数据库设计部分。不过我们的同事 Pramod Sadalage 和 Scott Ambler 合著了《数据库重构》，建议读者阅读。

当提到 DBA 时，我们指的是那些设计数据架构、编写代码访问数据并在应用中使用数据的人；编写在数据库中执行的代码，维护数据库并优化其性能的人；在故障发生时，确保数据库正常备份和恢复的人。通常 DBA 和开发人员是应用的核心构建者，他们应该紧密合作。

5.1 演进式数据库设计

数据库的演进式设计指开发人员能够根据需求的不断变化来构建数据库结构并使其演进。数据库模式是抽象的，和类的层次结构类似。当现实世界发生变化，这些变化需要反映在开发人员和 DBA 所建立的抽象当中。否则，抽象将逐渐脱离与现实世界的同步。

5.1.1 数据库模式演进

在继续使用传统工具（比如关系型数据库）的情况下，架构师如何构建支持演进的系统呢？数据库设计演进的关键在于数据库模式和代码演进。持续交付使得传统数据孤岛能够适应现代软件项目的持续反馈环。 开发人员必须以和代码变更相同的方式处理数据库结构的变更，它们必须是经过检验的、版本化的和增量的。

- **经过检验的**

  为了保证稳定性，DBA 和开发人员应该严格测试数据库模式的变更。如果开发人员使用了数据映射工具，例如对象关系映射器（ORM），那么为了使映射关系和数据库模式保持同步，他们应该考虑为此添加适应度函数。

- **版本化的**

  开发人员和 DBA 应该对数据库模式和那些使用它的代码一同进行版本控制。源代码和数据库模式是共生的，缺一不可。人为分离这两种必然耦合的事物的工程实践将导致低效。

- **增量的**

  和代码变更一样，变更数据库模式应该是渐进的，即随着系统的演进而增量地进行。现代工程实践往往使用自动化的迁移工具，从而避免手动更新数据库模式。

利用数据库迁移工具使得开发人员（或 DBA）能够对数据库进行小的增量变更，这一过程将作为部署流水线的一部分自动完成。从简单的命令行工具到复杂的、原始 IDE 的能力都涉及它。当开发人员需要变更某个数据库模式时，他们就会编写一小段 SQL 脚本，如示例 5-1 所示。

示例 5-1　简单的数据库迁移

```
CREATE TABLE customer (
       id BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1) PRIMARY KEY,
       firstname VARCHAR(60),
       lastname VARCHAR(60)
);
```

迁移工具将前面这段 SQL 脚本自动应用于开发人员的数据库实例中。如果开发人员之后发现忘了添加生日字段，他们可以创建一条新的迁移脚本来修改先前的架构，而不用修改先前的迁移脚本，如示例 5-2 所示。

示例 5-2　使用迁移工具为现有数据表添加生日字段

```
ALTER TABLE customer ADD COLUMN dateofbirth DATETIME;

--//@UNDO

ALTER TABLE customer DROP COLUMN dateofbirth;
```

在示例 5-2 中，开发人员通过添加新字段来修改现有数据库模式。有些迁移工具支持**撤销**，使得开发人员能够轻松地在数据库模式的版本中前进和后退。例如，某个项目的代码库处于 101 版本，但它需要退回到 95 版本。对源代码来说，开发人员只需要从版本工具中签出 95 版本即可。但如何保证数据模式对 95 版本的代码是正确的呢？如果他们使用的迁移工具具有撤销功能，那么数据库模式便能通过撤销退回到 95 版本，即反向执行每一条迁移脚本从而退回到任意版本。

然而，基于三个原因，大部分团队移除了撤销功能。首先，如果所有的迁移脚本都在，开发人员可以将数据库构建为他们所需要的版本，而不需要退回到先前某个版本。在示例中，开发人员可以从 1 版本到 95 版本来进行构建，从而恢复 95 版本。其次，为什么我们需要维护正向和反向两个版本的正确性呢？为了实现可靠的**撤回**，开发人员必须测试代码，有时会使测试负担加倍。最后，为了构建完整的**撤回**，有时会带来极大的挑战。设想迁移脚本删除了某个表，那么在进行**撤销**操作时，迁移脚本如何保留数据呢？

一旦开发人员执行了迁移脚本，这些变更就被视作不可变的，因为变更发生在复式记账之后。设想程序员丹妮尔执行了示例 5-2 中的迁移脚本，该迁移脚本是某个项目的第 24 个脚本。不久后，丹妮尔发现根本不需要生日字段（`dateofbirth`）。她可以删除该迁移脚本，最终表中将不会有生日字段。然而，在丹妮尔的迁移脚本生效期间编写的任何代码仍会假设生日字段的存在。如果由于某些原因，项目需要退回到某个中间点（例如要修复一个 bug），那么这些代码将无法再正常工作。为此她运行了一个新的迁移脚本来删除不再需要的字段。

数据库迁移将数据库模式和代码变更视为整体变更的一部分，这让 DBA 和开发人员能增量管理这些变更。将数据库变更整合到部署流水线的反馈环中，使得开发者可以将自动化和早期验证整合到项目的构建周期中。

5.1.2 共享数据库集成

下面着重介绍一个常见的集成模式——共享数据库集成，它以关系型数据库为数据共享机制，如图 5-1 所示。

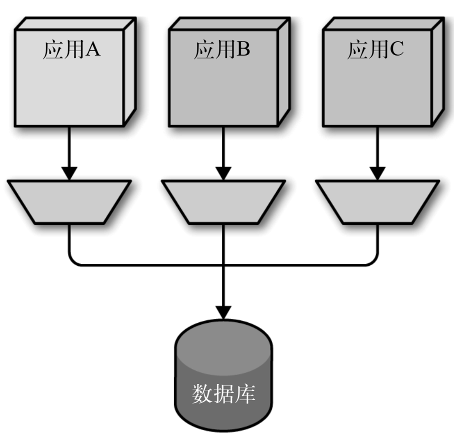


图 5-1：将数据库作为集成点

如图 5-1 所示，三个应用共享同一个关系型数据库。项目经常默认采用这种集成方式，为了方便管理，所有项目使用同一个关系型数据库，那为什么不在项目间共享数据呢？架构师很快就发现，使用数据库作为集成点僵化了所有共享项目的数据库模式。

当耦合的应用需要通过数据库模式变更来演进时会发生什么？如果应用 A 修改了数据库模式，可能破坏其他两个应用。幸运的是，正如《数据库重构》一书所讨论的，常用**扩展 / 收缩**重构模式来化解这种耦合。很多数据库重构技术通过在重构的过程中创建过渡阶段来避免时间问题，如图 5-2 所示。

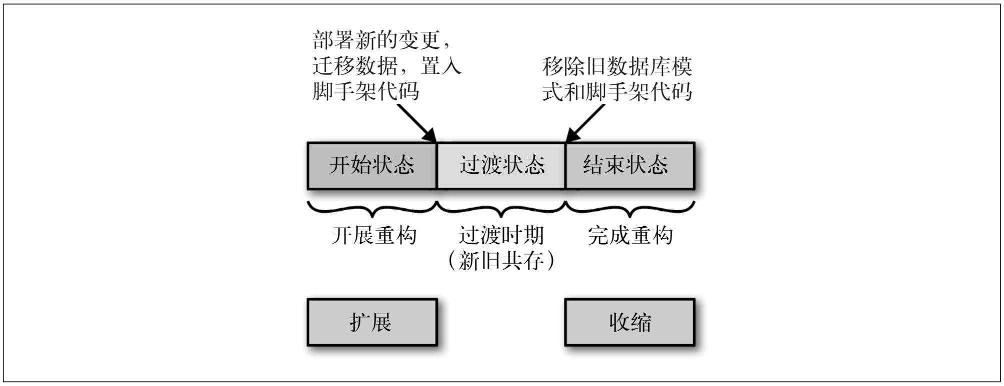


图 5-2：数据库重构的扩展 / 收缩模式

使用该模式，开发人员会设置开始状态和结束状态，它们会在转变过程中分别维持**新**、**旧**两种状态。这个过渡状态允许向下兼容，同时还给企业内的其他系统足够的时间来跟上变化。对于某些组织，这种过渡状态可持续几天到几个月。

下面是一个**扩展 / 收缩**模式在实践中的例子。考虑一个常见的演进式变更：Penultimate-Widgets 为了营销，需要将姓名（`name`）拆分成姓（`lastname`）和名（`firstname`）。为了这个变更，开发者经历了开始状态、扩展状态以及最终状态，如图 5-3 所示。

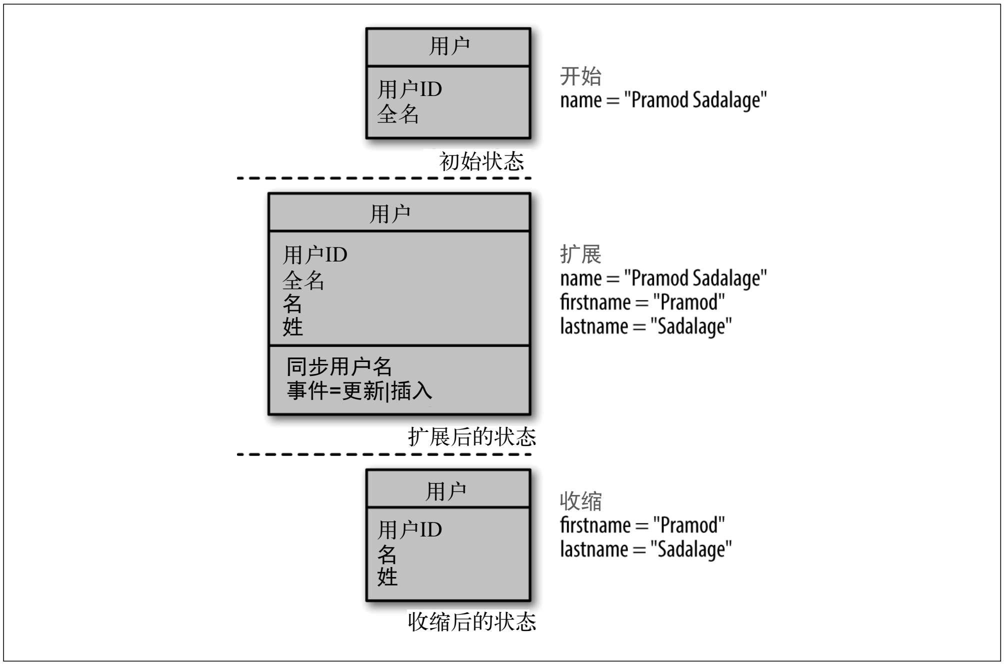


图 5-3：扩展 / 收缩重构的三个状态

如图 5-3 所示，全名是一个单独的字段。在转变过程中，PenultimateWidgets 的 DBA 必须维护两个版本以防止数据库中的集成点受到破坏。对于如何将 `name` 字段拆分成 `lastname` 和 `firstname`，有以下几种选项。

1. 选项 1：没有集成点和遗留数据

针对这种情况，开发人员不需要考虑其他系统，也不用管理已有的数据，因此他们可以直接添加新字段并删除旧字段，如示例 5-3 所示。

示例 5-3　没有集成点和遗留数据的简单情况

```
ALTER TABLE customer ADD firstname VARCHAR2(60);
ALTER TABLE customer ADD lastname VARCHAR2(60);
ALTER TABLE customer DROP COLUMN name;
```

对选项 1 来说，重构非常简单，DBA 做出相应的变更就行了。

2. 选项 2：存在遗留数据，但是没有集成点

在这样的场景中，开发者只需将已有数据迁移到新的字段，而不用担心任何外部系统。他们必须创建一个函数，从已有字段中提取相关信息来处理数据迁移，如示例 5-4 所示。

示例 5-4　不存在集成点的遗留数据迁移

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);
```

```
UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);
ALTER TABLE customer DROP COLUMN name;
```

这样的场景要求 DBA 提取和迁移已有数据，但其他方面很简单。

3. 选项 3：遗留数据和集成点同时存在

这种场景最为复杂，却也是最常见的。在外部系统依赖 `name` 字段的同时，开发人员需要将现有字段迁移到新的字段，而外部系统的开发人员无法在期望的时间内完成对新字段的迁移。示例 5-5 展示了适用于该场景的 SQL。

示例 5-5　存在遗留数据和集成点的复杂情况

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);

UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);

CREATE OR REPLACE TRIGGER SynchronizeName
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
BEGIN
  IF :NEW.Name IS NULL THEN
    :NEW.Name := :NEW.firstname||' '||:NEW.lastname;
  END IF;
  IF :NEW.name IS NOT NULL THEN
    :NEW.firstname := extractfirstname(:NEW.name);
    :NEW.lastname := extractlastname(:NEW.name);
  END IF;
END;
```

为了构建示例 5-5 中的过渡阶段，DBA 在数据库中添加了触发器。当其他系统向数据库添加数据时，触发器将数据从旧的 `name` 迁移到新的 `lastname` 和 `firstname` 字段中。这使得新系统可以访问相同的数据。同样，当新系统插入数据时，开发人员或 DBA 将 `lastname` 和 `firstname` 连接起来保存到 `name` 字段，这样其他系统依然可以访问格式正确的数据。

一旦其他系统修改其数据访问去使用新的结构（具有单独的 `lastname` 和 `firstname`），便可以执行**收缩**阶段来删除旧字段。

```
ALTER TABLE Customer DROP COLUMN name;
```

如果存在大量数据并且删除字段很耗时，那么 DBA 可以将该字段设置为“弃用”（如果数据库支持该功能）。

```
ALTER TABLE Customer SET UNUSED name;
```

在删除了旧字段之后，如果需要之前数据库模式的只读版本，那么 DBA 可以添加一个函数字段，这样便保留了对旧字段的只读访问。

```
ALTER TABLE CUSTOMER ADD (name AS
          (generatename (firstname,lastname)));
```

正如每种情况所展现的，DBA 和开发人员可以利用数据库的固有功能来构建可演进的系统。

扩展 / 收缩模式是平行变更模式的一个子集，广泛用于安全地对接口执行向下不兼容变更。

5.2 不当的数据耦合

数据和数据库是大多数现代软件架构中不可或缺的部分，如果开发人员忽略这一关键因素，在尝试演进架构时将会遭遇挫折。

在很多组织中，数据库和 DBA 面临特别的挑战，由于各种原因，相比传统的软件开发，他们的工具和工程实践都非常过时。例如，相对于开发人员的 IDE 而言，DBA 日常工作所用的工具非常原始。那些对开发人员而言很常见的功能 DBA 却不了解，例如重构支持、容器外测试、单元测试、模拟和打桩等。

DBA、数据库厂商和工具选型

为什么数据的工程实践远远落后于软件开发？DBA 和开发人员有很多相同的需求，例如测试、重构等。但是，在开发者工具不断发展的时候，数据领域没有发生同等的创新。并不是没有合适的工具，市面上的一些第三方工具能更好地支持工程，但销量并不好，这是为什么？

数据库厂商在他们和消费者之间建立了一种有趣的联系。例如，某数据库厂商的 DBA 对该厂商的奉献程度几乎无以复加，因为 DBA 的下一份工作至少一部分是来自于他是该厂商认证的 DBA，而不一定是源于其当前的工作。因此，数据库厂商在世界各地的企业中拥有一支秘密军队，他们效忠于数据库厂商而不是其雇主。在这种情况下，DBA 会忽略那些不是来自数据库厂商的工具和开发组件。其结果便是工程实践的创新水平停滞不前。

DBA 将数据库厂商视为宇宙中光和热的来源，而不在意宇宙中其他暗物质。相比开发工具，该现象使得数据库工具的发展停滞不前。因此，由于开发者和 DBA 之间不共享通用的工程技术，他们之间的差距变得越来越大。说服 DBA 开始采用持续交付实践，迫使他们开始使用新工具，将疏远他们和数据库厂商之间的关系，而这是厂商们试图避免的。

然而开源数据库和 NoSQL 数据库的普及动摇了数据库厂商的霸主地位。

5.2.1 二阶段提交事务

当架构师讨论耦合时，对话通常围绕类、库和技术架构的其他方面展开。但是，在大多数项目都有产生耦合的其他途径，比如事务。

事务是耦合的一种特殊形式，因为事务性行为不会在传统的、以技术架构为中心的工具中出现。架构师能够轻松地通过各种工具确定类的传入和传出耦合。但是，确定事务上下文范围要困难得多。正如数据库模式的耦合会妨碍演进一样，事务性耦合以具体的方式将各个组成部分绑定在一起，使得难以演进。

出于各种各样的原因，事务出现在业务系统中。第一，业务分析师青睐“事务”这个概念（一种在某些情况下**暂停世界**的简单操作），而忽略技术上的挑战。在复杂系统中很难进行全局协调，而事务就是某种形式的全局协调。第二，我们通常能从事务边界中看出在实现层面上业务概念是如何耦合在一起的。第三，DBA 可能管理事务上下文，这导致在技术架构中拆分数据来耦合将难以协调。

在尝试将事务性很强的系统转换为不当的架构模式（如微服务）时，开发者会将事务视为耦合点，从而带来了繁重的解耦负担。基于服务的架构对服务边界和数据分区的要求不那么严格，更适合事务性系统。4.3.4 节讨论了这些架构的相关差异。

第 1 章和第 4 章讨论了架构量子边界概念的定义，即架构可部署的最小单元，区别于传统内聚思维，它封装了依赖组件，例如数据库。由数据库创建的绑定强于传统的耦合，因为事务边界通常定义了业务流程的运作方式。架构师有时在构建架构时错误地将架构粒度设计得比业务本身的粒度还小。例如，微服务架构不适合事务性很强的系统，因为微服务的架构量子太小。基于服务的架构往往更合适，因为它对架构量子大小没有严格要求。

架构师必须考虑应用的所有耦合特征，其中包括类、包 / 命名空间、库、框架、数据库模式，以及事务上下文。在架构演进时，忽视其中任一维度（或维度间的交互）都将产生问题。在物理学中，**强核力**将原子束缚在一起，它是迄今为止所我们了解的最强力量之一。事务上下文对架构量子来说，就好像强核力一样。

> 数据库事务好像强核力一样，将架构量子结合在一起。

虽然系统通常无法避免事务，但架构师应尽可能地限制事务上下文，因为它们形成了紧密的**耦合结**，妨害了在互不影响的情况下变更组件或服务的能力。更重要的是，架构师在考虑架构变更时应将事务边界考虑在内。

如第 8 章将要讨论的，在将单体架构迁移到某种更精细的架构时，应该从分离少量较大的

服务开始。当构建一个全新的微服务架构时，开发人员应该尽量限制服务和数据上下文的大小。然后，不要仅按字面意思来理解**微服务**，对每个服务而言，小并不是必需的，能捕获有用的限界上下文才是关键。

在重建某个现有的数据库模式时，通常很难达到合适的粒度。很多企业级 DBA 花费了数十年的时间将数据库模式整合到一起，他们对这种逆向操作不感兴趣。通常，那些支撑业务的必要事务上下文定义了开发者所能构建服务的最小粒度。虽然架构师想构建更小的粒度，但如果粒度大小和数据问题不匹配，那么他们的行为将导致不适当的耦合。如果架构和开发人员尝试解决的问题在结构上相冲突，那么该架构就破坏了元工作，详见 6.4 节。

5.2.2　数据的年龄和质量

大型企业的另一种机能障碍是对数据和数据库的痴迷。我们经常听到 CTO 们说："我不大关心应用本身，因为应用的寿命是短暂的，**但数据模式是宝贵的，因为它永久有效。**"虽然模式变更的频率确实比代码要低，但数据库模式仍然是对现实世界的抽象。现实世界在不断变化，因此那些确信数据库模式不会变化的 DBA 们事实上忽视了现实。

但是如果 DBA 从来不修改数据库模式，以重构数据库，那么他们如何进行变更来适应新的抽象呢？很不幸，通常 DBA 通过**添加另一张连接表**来扩展数据库模式定义。与其冒着破坏现有系统的风险更改数据库模式，他们往往添加一张新的数据表，并通过关系型数据库原语将其与原始数据表关联起来。虽然这样做短期内有效，但是它混淆了真实的根本抽象，因为在现实世界中，一个实体是通过多个事物表现的。随着时间推移，那些几乎不懂重构数据库模式的 DBA 通过拜占庭分组和聚束策略构建出了日渐僵化的模式。如果 DBA 不重构数据库，那么他们将无法保留宝贵的企业资源，反而创建出数据库模式的具体残留，它们通过连接表重叠在一起。

遗留数据的质量可能带来另一个大的问题。通常数据会历经多代软件，每代软件都有其特殊的持久化需求，这样导致的最好结果是数据不一致，最坏的结果则是垃圾数据。在许多方面，尝试将每份残留的数据与过去的架构耦合，会迫使我们寻求一些变通的方法来保持系统正常工作。

在尝试构建演进式架构前，确保开发人员能够对数据模式和质量进行演进。糟糕的结构需要重构，并且 DBA 应该采取必要的措施来保证基本的数据质量。我们倾向于尽早解决这些问题，而不是建立精心设计的、持续的机制来反复地处理这些问题。

遗留的数据库模式和数据具有价值，但它们也妨碍了系统的演进能力。架构师、DBA 和业务代表需要展开坦诚的对话，讨论哪个对组织更有**价值**，是永久地保存遗留数据还是进行演进式变更的能力。我们应识别真正有用的数据并将其保留下来，将旧数据作为参考但不将其纳入演进式开发的主流。

拒绝重构数据库模式或删除旧数据会使架构耦合到过去，这将导致重构难以进行。

5.3　案例研究：PenultimateWidgets的路由演进

PenultimateWidgets 决定在页面间实施新的路由方案，为用户提供导航面包屑跟踪。这样做意味着要修改现有的页面路由方式（使用内部框架）。那些需要实施新路由机制的页面需要更多的上下文和数据，例如源页面、工作流状态等。

在路由服务中，PenultimateWidgets 目前有一张单独的数据表在处理页面路由。在新版本中，开发人员需要更多信息，因此表的结构会更复杂。设想实施新路由的起始状态，如图 5-4 所示。

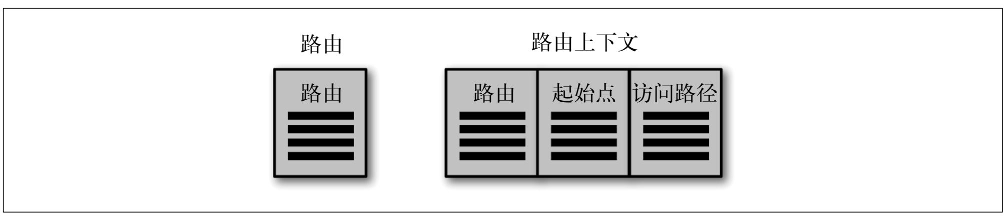


图 5-4：实施新路由的起始状态

由于不同的业务单元的工作节奏不同，因此 PenultimateWidgets 的页面不会同时实施新的路由。因而路由服务必须支持新、旧两个版本。第 6 章将介绍路由服务是如何处理的。在本例中，我们会在数据层面处理相同的场景。

使用“扩展 / 收缩”模式，开发人员可以创建新的路由结构并使其通过服务调用可用。在内部，两个路由表都有和 `Route`（路由）字段相关的触发器，因此对其中一张表的变更将自动复制到另一张表中，如图 5-5 所示。

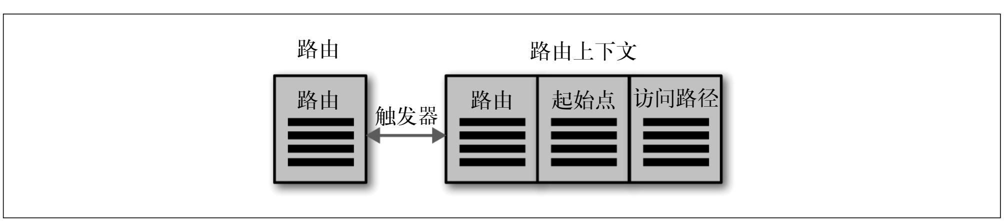


图 5-5：在过渡状态中服务会同时支持两个版本的路由

如图 5-5 所示，如果开发者需要旧的路由服务，那么该服务将支持这两种 API。本质上，该应用现在支持两个版本的路由信息。

当不再需要旧的服务时，路由服务的开发人员便能删除旧的服务和触发器，如图 5-6 所示。

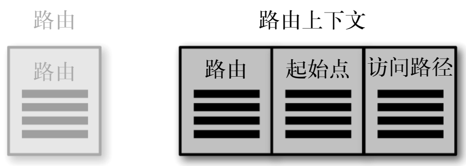


图 5-6：路由表的最终状态

在图 5-6 中，所有服务都已经迁移到了新的路由服务，开发人员可以删除旧的服务。这和图 5-2 所示的工作流一致。

只要开发人员应用正确的工程实践（持续集成、代码版本控制等），数据库便可以和架构一同演进。这种可以轻易地变更数据库模式的能力至关重要，因为数据库代表了对现实世界的某种抽象，而现实世界的变化难以预料。虽然数据抽象比行为更抵制变化，但它仍然需要演进。在构建演进式架构时，架构师必须将数据看作首要问题。

重构数据库是 DBA 和开发人员需要掌握的重要技能。对很多应用而言，数据是根本。为了构建可演进的系统，开发人员和 DBA 需要在采用其他现代工程实践的同时尝试那些有效的数据实践。

第6章

构建可演进的架构

至此，我们已经分别介绍了演进式架构的三个主要方面——适应度函数、增量变更和适当的耦合，是时候将三者联系起来了。

我们讨论的很多概念并不新鲜，只是以新的视角来审视。例如，测试已经存在了很多年，但并不是以适应度函数的形式强调架构验证的。**持续交付**定义了部署流水线，演进式架构展示了这一能力的实际效用。

很多组织青睐持续交付实践，将其作为提升软件开发工程效率的手段。然而，我们更进了一步，运用这些能力构建更复杂的架构，它能随着现实世界不断演进。

那么开发人员如何在新、旧项目中利用这些技术呢？

6.1 演进机制

架构师可以运用这些技术通过下面三步来构建演进式架构。

6.1.1 识别受演进影响的架构维度

首先，架构师必须确定在演进中他们想保护的架构维度。其中一定包含技术架构，通常还有数据设计、安全、可伸缩性和其他一些他们认为重要的特征。该过程会涉及组织内部其他相关团队，包括业务、运营、安全和其他受影响的团队。逆康威时刻（1.5 节介绍过）对此很有帮助，因为它鼓励组织多角色团队。基本上，这是架构师在项目初期确定需要支持的架构特征的常规工作。

6.1.2 为每个维度定义适应度函数

单个架构维度通常包括多个适应度函数。例如，为了保证代码的架构特征，架构师通常将一系列代码衡量指标构建到部署流水线中，例如避免组件循环依赖。架构师通过轻量级的方式记录那些需要持续关注的架构维度，例如 wiki。接着，针对每个维度，他们确定在演进过程中可能出现错误行为的部分，最终定义出适应度函数。适应度函数可以自动运行或手动触发，并且在某些情况下需要设计得更加巧妙。

6.1.3 使用部署流水线自动化适应度函数

最后，架构师必须在项目中推进增量变更，在部署流水线中定义不同阶段来执行适应度函数并管理部署实践，例如环境准备、测试和其他 DevOps 问题。增量变更是演进式架构的引擎，它让我们可以通过部署流水线主动验证适应度函数，并通过高度自动化隐藏一些单调的任务，例如无感知部署。生产周期是在持续交付中衡量工程效率的指标。在支持演进式架构的项目中，开发人员的职责之一就是保持良好的生产周期。生产周期是增量变更的重要部分，因为其他很多度量指标亦来源于此。例如某架构中新版本的发布速度和其生产周期成正比。换句话说，一旦项目的生产周期延长，那么它将使项目交付新版本的速度减慢，进而影响演进能力。

虽然是在新项目初期定义架构维度和适应度函数，但对新项目和已有项目而言，它都是一个持续的活动。开发人员无法预料所有事情，因此软件会受到未知的未知问题的困扰。在构建软件的过程中，架构的某些部分有时会显露出不好的迹象，构建适应度函数能阻止问题进一步恶化。虽然有些适应度函数会在项目初期自然显现，但还有一些适应度函数在架构经受压力时才会显现。架构师尤其需要注意那些非功能性需求被破坏的情况，并通过适应度函数更新架构来避免可能出现的问题。

6.2 全新的项目

为新项目构建演进能力远比改造已有项目容易。首先，开发人员可以立即进行增量变更，在项目初期构建部署流水线。在编写代码前，确定和设计适应度函数将更容易，也便于将适应度函数纳入系统，因为在初期项目脚手架就已经搭建完成了。其次，架构师无须理清那些蔓延到现有系统的耦合点。而且，架构师可以让度量和其他验证机制就位，从而保证项目变更时架构的完整性。

如果开发人员选择了正确的架构模式和工程实践来构建演进式架构，那么新项目在处理意外变更时会更容易。例如，微服务架构以其极低的耦合和极强的增量变更能力成为了显而易见的候选方案（其流行程度也是一个因素）。

6.3 改良现有架构

赋予现有架构演进能力取决于三个因素：组件耦合度、工程实践成熟度，以及开发人员构建适应度函数的难易程度。

6.3.1 适当的耦合和内聚

组件间的耦合很大程度上决定了技术架构的演进能力。然而，如果数据模式死板、僵化，即便最具演进性的技术架构也注定会失败。清晰解耦的系统易于演进，充满耦合的系统则会妨碍演进。想构建出真正可演进的系统，架构师必须考虑架构中所有受影响的维度。

除了技术层面的耦合，架构师还必须考虑和保护系统中组件的功能内聚。当从一种架构迁移到另一种架构时，功能内聚性决定了组件重构后的最终粒度。这并不意味架构师可以随心所欲地分解组件，而是说基于特定的问题上下文，组件的大小应该是适当的。例如，有些业务问题相较其他问题耦合度更高，比如有着大量事务的系统。试图构建极其解耦的架构与这类问题相左，因此是徒劳的。

选择架构前，需要理解面临的业务问题。

虽然这个建议看似多余，但是仍然有团队选择最炫目的新架构模式，而不是最合适的架构。在一定程度上，选择架构要基于对业务问题和物理架构的综合理解。

6.3.2 工程实践

工程实践对定义架构的可演进性至关重要。虽然持续交付实践无法保证架构能实现演进，但它依然不可或缺。

为了追求效率，很多团队着手改进工程实践。一旦这些实践增强了，它们便能为更高级的能力（例如演进式架构）提供基础。因此，构建演进式架构能够促进效率提升。

很多公司处于新、旧实践的过渡阶段。它们可能已经实现了一些相对容易的实践，例如持续集成，但是仍然需要大量手工测试。虽然这些手工测试会延长生产周期，但在部署流水线中包含一些手动阶段很重要。第一，这样会将应用构建的每个阶段都置入部署流水线中。第二，随着团队逐步将更多部署工作自动化，手动阶段也会实现自动化，不再中断部署过程。第三，阐明每个阶段有助于我们更好地理解构建的各个手工部分，创造更好的反馈环并推动改进。

通常，构建演进式架构的最大障碍是棘手的运维工作。如果开发人员无法轻松地部署变更，反馈环的各个部分都会受阻。

6.3.3　适应度函数

适应度函数是演进式架构的保护层。如果架构师围绕特定的架构特征构建系统，那么这些特征可能和可测试性形成正交关系。幸运的是，现代工程实践极大地提升了可测试性，使得过去难以验证的架构特征能够自动被验证。在演进式架构中，这将占大部分工作，而适应度函数使那些以前不同的关注点得以平等对待。

我们希望架构师将各种架构验证机制视为适应度函数，包括那些临时考虑的事情。例如，很多架构中都含有与伸缩性相关的服务级别协议（SLA）及相应测试。它们还具有围绕安全需求的业务规则及附带的验证机制。架构师往往区分它们，但两者的目的是一致的，即验证架构的某些特性。将所有架构验证视为适应度函数，在定义自动化和其他有用的协同交互时能够达到更好的一致性。

> **重构与重建**
>
> 开发人员有时会赋予那些听起来不错的术语更广泛的含义，例如“重构”。正如 Martin Fowler 所定义的那样，“重构”是在不改变既有计算机代码外部行为的情况下，对其进行调整的过程。对很多开发人员来说，“重构”已经成为了“变更”的代名词，但它们之间存在关键的不同。
>
> 团队重构架构是极其罕见的。相反，他们会重建架构，即对结构和行为进行实质性的改变。某种程度上，架构模式的存在是为了使特定的架构特征在应用程序中成为主要特征。切换架构模式需要切换优先级，因此这并不是重构。例如，为了伸缩性，架构师可能选择采用 EDA。如果团队切换到不同的架构模式，架构很可能无法保持同等的伸缩能力。

6.3.4　关于商业成品软件

在许多组织中，开发人员并不负责构建系统的所有部分。COTS（商业成品软件）和套装软件在大型企业中非常普遍，它们给架构师构建可演进的系统带来了挑战。

COTS 必须随着企业中的其他应用一同演进，然而这些系统没有很好地支持演进。

- 增量变更

  可悲的是，大多数商用软件的自动化和测试都落后于行业标准。架构师和开发人员必须经常隔离集成点并尽可能地构建测试，并将整个系统视为黑盒。在实施敏捷性时，COTS 在部署流水线、DevOps 及其他现代实践方面给开发团队带来了很多挑战。

- 适当的耦合

  套装软件经常在耦合上出错。通常，这类系统是不透明的，开发者使用预定义的 API 进行集成。这些 API 不可避免地会遭遇 7.1.3 节将介绍的反模式的问题，它们给开发人员些许（但不太够的）灵活性来完成重要的工作。

- 适应度函数

  在赋予系统演进能力的征途上，最大的障碍可能是为套装软件添加适应度函数。通常，这类软件不会暴露太多内部细节，因此难以进行单元测试和组件测试，只能诉诸于基于行为的集成测试。但这类测试并不理想，因为它们粒度太大，必须在复杂的环境中运行并覆盖大部分系统行为。

> 我们应努力使集成点保持一定的成熟度。如果无法做到这一点，那么你需要意识到系统的某些部分将更易于演进。

不透明的数据库系统是套装软件供应商引入的另一个令人担忧的耦合点。最好的情况是，套装软件全权管理数据库，通过集成点暴露选定的数据。最坏的情况是，供应商数据库就是对外集成点，使 API 两端的变更变得极其复杂。在这种情况下，架构师和 DBA 必须对套装软件中的数据库有控制权才可能实现演进。

如果不可避免地受到套装软件的困扰，架构师应该尽可能地构建强大的适应度函数，并使其自动化地运行。由于对套装软件内部的访问受限，测试技术在这种情况下并不太理想。

6.4 架构迁移

很多公司最终会从一种架构迁移到另一种架构。例如，在公司刚开始构建 IT系统时，架构师会选择易于理解的架构模式，通常是分层的单体架构。随着公司发展，架构变得不堪重负。出于架构思维中以领域为中心的转变思路（参考 4.3.4 节中的“微服务”部分），最常采用的一种架构迁移路径是从单体迁移到某种基于服务的架构。很多架构师对高度可演进的架构感兴趣，并将其作为迁移的目标，但迁移通常非常困难，主要由现有的耦合所致。

当架构师想迁移架构时，通常会考虑类和组件间的耦合特征，但可能忽略其他很多影响演进的维度，例如数据。和类之间的耦合一样，也存在事务性耦合，而且在重建架构时难以消除。当尝试将现有模块分解得更小时，这些额外的耦合点带来了巨大的负担。

很多资深开发人员年复一年地构建相同类型的应用，因单调而厌倦。于是，很多开发人员倾向于编写框架，而不是使用现成的框架来构建应用，这便是所谓的“元工作比工作更有趣”。工作是无趣、平凡且重复的，而构建新事物则令人兴奋。

这体现在两个方面。第一，很多资深开发人员编写底层代码给他人使用，而不是利用已有的软件（通常是开源）。我们的某个客户曾处于技术前沿。他们用 Java 构建了自己的应用服务器、Web 框架以及几乎所有的基础设施。有一次，我问他们是否还构建了自己的操作系统，他们说："没有。"我继续问道："为什么不呢？你们从零构建了其他所有东西。"

经过反思，他们发现公司缺乏某些能力。然而，当开源工具可以提供这些能力时，他们已经拥有了钟爱的自研基础设施。由于方法上存在细微差别，他们决定坚持使用自研的工具，而不是标准技术。十年后，他们最优秀的开发人员忙于维护这些工具、修复应用服务器、为 Web 框架添加新特性和其他杂事。他们长期困于维护，无暇创新以构建更好的应用。

第二，架构师无法对"元工作比工作更有趣"综合征免疫，这种综合征表现为采用时髦但并不合适的架构，例如微服务。

> 不要仅仅因为元工作有趣而构建架构。

6.4.1 迁移步骤

架构师在从过时的单体应用向更现代的、基于服务的架构迁移的过程中面临巨大挑战。有经验的架构师发现应用中存在大量耦合点，在分解代码时，首要任务便是理解它们之间的联系。当分解单体应用时，架构师必须考虑耦合和内聚，寻求两者间的平衡。例如，微服务架构中最严格的约束便是数据库必须置于服务的限界上下文内。当分解单体应用时，即便可以将类分解得足够小，但想把事务上下文也分解到相似的程度仍难如登天。

> 在重建架构时，需要考虑所有受影响的维度。

很多架构师最终从单体应用迁移到基于服务的架构。考虑如图 6-1 所示的迁移起始状态。

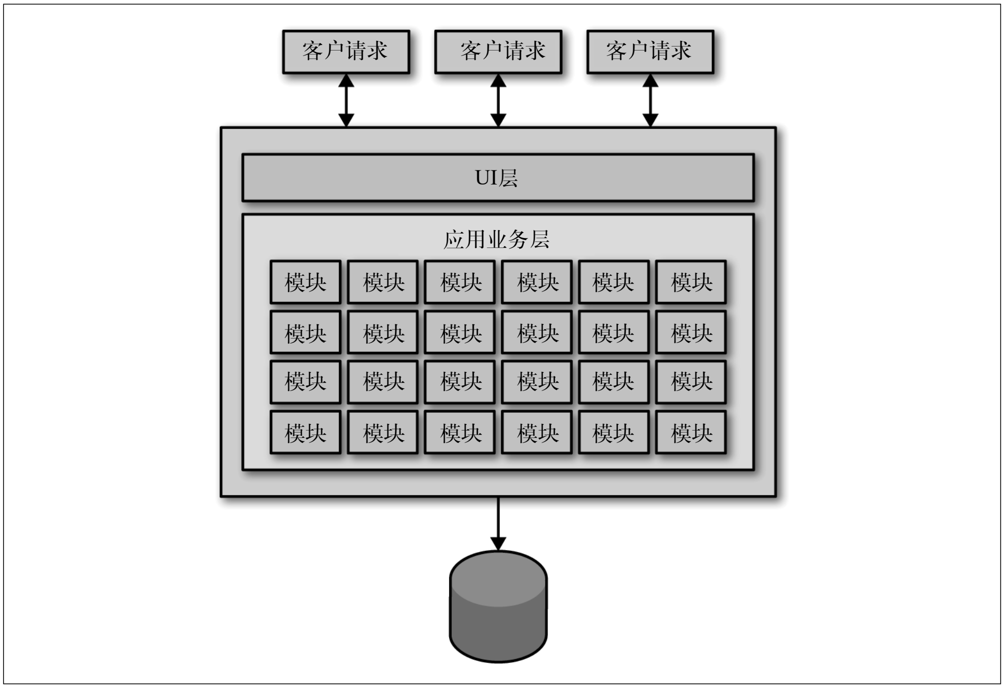


图 6-1："共享一切"的单体架构是迁移的起点

在新项目中构建细粒度服务很简单，但是在已有的项目中构建则很困难。因此，我们如何将图 6-1 所示的架构迁移到基于服务的架构（见图 6-2）呢？

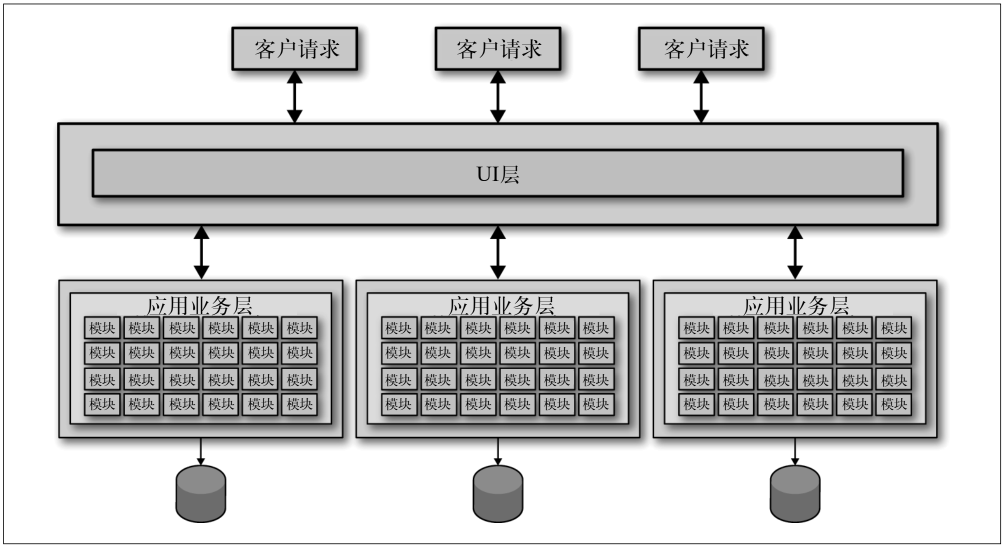


图 6-2："尽可能少共享"的、基于服务的架构是迁移目标

实施从图 6-1 到图 6-2 所示的架构迁移会带来一系列挑战：服务粒度、事务边界、数据库问题以及如何处理共享组件。架构师必须清楚实施该迁移的原因，并且确保不是盲目地赶时髦。将架构划分为领域，加上更好的团队结构和运维的隔离，会使增量变更更容易，这是演进式架构的关键组成之一，因为工作的重点和实际的产出是相互匹配的。

在分解单体架构时，确定正确的服务粒度是关键。创建大型服务能够缓解一些问题，例如事务上下文和服务编制，但是无法将单体应用分解到更小的粒度。粒度过小的组件则会导致过多的服务编排和消息通信所带来的开销以及服务间的相互依赖。

为了迈出架构迁移的第一步，开发人员要定义新的服务边界。团队可以通过多种划分方式将单体应用分解成服务。

- 业务功能分组

  企业可能有清晰的业务划分直接对应于 IT 能力。模仿当前业务沟通层级构建的软件无疑应验了康威定律（参见 1.5 节）。

- 事务边界

  许多业务需要依附于大量事务边界。当分解单体应用时，架构师经常发现事务耦合是最难解开的，正如 5.2.1 节所讨论的。

- 部署目标

  增量变更使得开发人员可以按照不同的计划有选择地发布代码。例如，相比库存部门，市场部门可能希望更新频率更高。如果运维准则非常重要，例如发布速度，那么围绕运维问题划分服务是合理的。类似地，系统的某个部分可能对运维特征有极致的要求（例如伸缩性）。围绕运维目标划分需求使得开发人员能够（通过适应度函数）跟踪服务的健康状态和其他运维服务指标。

较大的服务粒度意味着微服务中许多固有的协调问题都不会存在，因为服务越大，单个服务所包含的业务上下文就越多，但同时操作难度也越大（又一个架构权衡的例子）。

6.4.2 演进模块间的交互

迁移共享模块（包括组件）是开发人员面临的另一个常见的挑战。设想图 6-3 所示的结构。

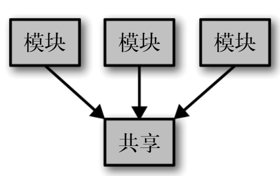


图 6-3：有着输入 / 输出耦合的模块

在图 6-3 中，三个模块共享相同的库。然而，架构师需要将这三个模块拆分到单独的服务中。那他要怎样维持这些依赖呢？

有时候可以清晰地拆分模块，分别保存不同模块所需要的功能，如图 6-4 所示。

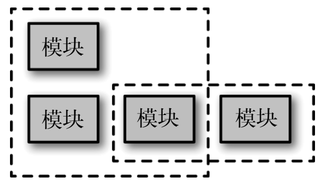


图 6-4：有着相同依赖的模块

在图 6-4 中，所有模块都需要红色模块（即中间的模块）。运气好的话，开发人员能把模块功能从中间一分为二，将共享模块拆分成各个依赖所需的相关部分，如图 6-5 所示。

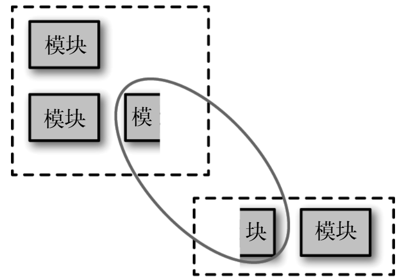


图 6-5：拆分共享的依赖

然而，更可能的是无法轻易切分共享模块。在这种情况下，开发人员可以将模块提取为共享的库（例如 JAR、DLL、gem 或其他组件机制），然后在需要的地方使用它，如图 6-6 所示。

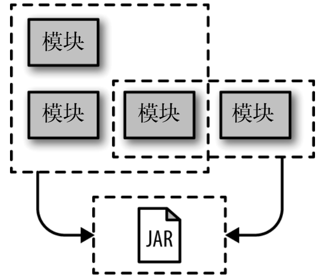


图 6-6：通过 JAR 文件共享依赖

共享就是耦合的一种形式，在微服务架构中这是非常不可取的。我们用复制作为替代方案来避免使用共享的库，如图 6-7 所示。

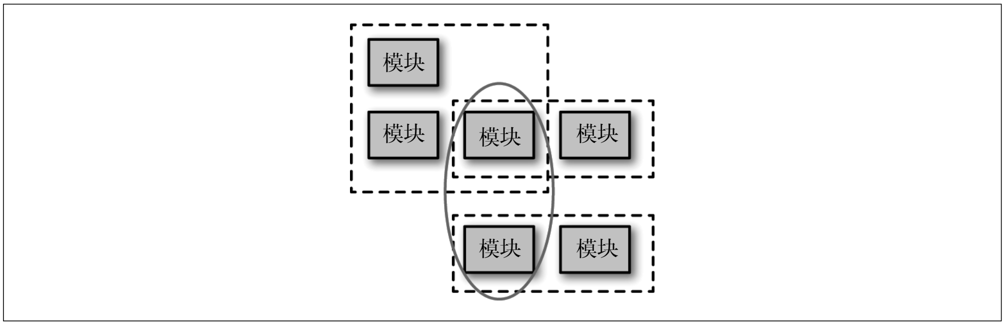


图 6-7：复制共享的库以消除耦合点

在分布式环境中，开发人员可以使用消息或服务调用来实现相同形式的共享。

当开发人员确定了正确的服务划分，下一步便是**分离**业务层和 UI。即便在微服务架构中，UI 也会回归到单体，毕竟在某些情况下开发人员必须展示统一的 UI。因此，开发人员通常会在迁移早期分离 UI，在界面组件和后端服务间构建映射代理层。在分离 UI 时，还会构建防腐层来将户界面的变更和架构变更隔离开。

接下来是**服务发现**，它让服务能够相互查找和调用。最终，架构将由必须相互协调的服务所组成。通过尽早地构建服务发现机制，开发人员可以从容地迁移系统中需要变更的部分。开发人员经常将服务发现构建成一个简单的代理层，每个组件调用代理，然后代理再将请求映射到指定的实现。

> 计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决，除非该问题是由间接层太多导致的。
>
> ——Dave Wheeler 和 Kevlin Henney

当然，开发人员增加的间接层越多，服务间的导航就会变得越复杂。

在将应用从单体架构迁移到基于服务的架构时，架构师必须注意现有系统中模块的连接方式。不成熟的划分方式会带来严重的性能问题。单体应用中的连接点成为了集成架构连接，伴随着延时、可用性和其他问题。因此，与其一次性完成整个迁移，更加实用的方法是循序渐进地将单体应用分解成服务，关注事务边界、结构耦合和其他固有特征来创建若干个架构重建迭代。首先将单体应用分解成几大块，修复集成点，然后清理和重复这个过程。逐步迁移是微服务领域的首选迁移方式。

> 当从单体应用迁移时，首先构建少量大型服务。
>
> ——Sam Newman，《微服务设计》[3]

注 3：此书已由人民邮电出版社出版，详见 http://www.ituring.com.cn/book/1573。——编者注

接下来，开发人员从单体应用中分离选中的服务并修复集成点。适应度函数在这里能起到关键作用，开发人员应构建适应度函数以确保新引入的集成点不会改变已有行为，并添加消费者驱动的契约。

6.5 演进式架构构建指南

本书使用了一些生物学隐喻，下面就是一个。人脑的进化并不是在理想的纯净环境中进行的，其能力也从未精心雕琢过。相反，大脑的每一层都基于其下方的原始层。人类大部分核心自主行为由大脑的各个部分主控，例如呼吸、饥饿等，这与爬行动物的大脑没什么区别。进化过程并没有将核心机制整体更换，而是在顶部构建新的层。

在大型企业中，软件架构遵循类似的模式。与其重建各项能力，大多数企业会努力适应现有的一切。尽管我们喜欢在纯净的理想环境中讨论架构，但现实世界往往展现出相反的混乱状态，技术债、优先级冲突和有限的预算很常见。在大型企业中，架构正如人脑一样：底层系统依旧处理着关键的业务细节，但也伴随着过去的包袱。企业不愿意放弃还在工作的系统，这导致集成架构的挑战不断升级。

赋予现有架构演进能力极具挑战。这是因为，如果开发人员从未将架构构建得易于变更，那么演进能力便不太可能自然地出现。架构师无法轻松地将大泥团转变成现代微服务架构，无论他多么有天赋。然而通过为项目增添一些灵活性，便能在不改变其整体架构的情况下改善项目。

6.5.1 去除不必要的可变性

稳定性是持续交付的目标之一，即在已知运行良好的系统上进行构建。构建不可变的基础设施这一现代 DevOps 视角体现了这一目标。第 1 章讨论了软件开发环境中的动态平衡，没有什么比软件所依赖的基础变迁更明显的了。随着开发人员更新功能、发布服务包、调整软件，软件系统在持续变化。例如操作系统，它们在持续变化着。

通过用**不可变的基础设施**取代**雪花服务器**，现代 DevOps 在其领域内解决了动态平衡的问题。运营人员手动创建雪花服务器，并手动完成将来所有的维护。Chad Fowler 在其博客“Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Compoents”中提出了“不可变的基础设施”。不可变的基础设施是指完全以程序的方式定义的系统。所有对系统的变更都必须通过代码完成，而不是修改运行中的操作系统。因此，从运维的角度来看，整个系统是不可变的，一旦开始运转，就不会发生变化。

虽然不可变性听起来与演进性背道而驰，但恰恰相反。软件系统由成千上万个动态部分组成，它们相互依赖、紧密联系在一起。然而当某个部分发生变化时，开发人员仍然努力应对各种意外。通过锁定意外变更的可能性，能更有效减少使系统变得脆弱的因素。开发人

员力图用常量代替代码中的变量来减少变化的因素。DevOps 将其引入运维中，使得运维工作更加申明式。

不可变的基础设施遵循了我们所提倡的**去除不必要的可变性**这一思路。构建可演进的软件系统意味着尽可能地控制未知因素。想构建适应度函数来预测操作系统最新的服务包对应用的影响，实际上是不可能的。于是，开发人员在每次部署流水线执行时都会重新构建基础设施，从而尽可能积极地捕获具有破坏性的变更。如果开发人员可以移除已知的底层、可更换的组件，例如操作系统，那么就可以减轻测试的负担。

架构师能通过各种途径将可变的事务变成常量。很多团队将“不可变的基础设施”这一理念延伸到了开发环境。开发人员经常声称“这在我的电脑上是正常的”。如果所有开发者使用完全相同的镜像，大量不必要的变量将不复存在。例如，大部分开发团队通过仓库自动更新开发依赖库，但是工具（例如 IDE）的升级怎么办呢？通过将开发环境构建成不可变的基础设施，开发人员能始终在相同的基础上工作。

构建不可变的开发环境还能让我们在项目中使用有用的工具。结对编程是敏捷开发团队的常见实践，其中包括轮转结对，即每个团队成员定期更换结对对象，从几个小时到几天不等。然而，当昨天使用的工具今天却找不到时非常令人沮丧。通过构建开发者系统的单一源，可以轻松地向所有系统添加有用的工具。

雪花服务器的危害

在某个知名博客上有这么一则故事，“噩梦：一个 DevOps 的警世寓言”，它可以作为雪花服务器的警世寓言。某个金融服务公司过去有个名为“PowerPeg”的算法，专门用于处理交易详情，但是那些代码多年没有用过了。然而，开发人员并没有删除这些代码。其功能开关处于关闭状态。由于监管需求的变化，开发人员实现了名为“SMARS”的新交易算法。但为了偷懒，他们决定在 SMARS 的代码实现中沿用 PowerPeg 的功能开关。2012 年 8 月 1 日，开发人员在七台服务器上部署了新代码。不幸的是，系统一共有八台服务器，其中一台服务器没有更新。当他们开启了 PowerPeg 功能开关后，其中七台服务器开始卖出，而剩下的那台开始买进！开发人员意外地设置了最坏的交易场景——它们自动地高买低卖。他们认为这是新代码惹的祸，因此开发人员在那七台服务器上回滚了新代码，但还是无法关闭功能开关，这意味着 PowerReg 现在在所有服务器上运行。最终他们用了 45 分钟使系统恢复正常，造成了高达 4 亿美元的损失。所幸某个天使投资拯救了他们，因为这家公司的价值不止如此。

这个故事强调了未知的可变性可能带来的问题。复用旧的功能开关是鲁莽的行为，功能开关的最佳实践是在其目的达成后尽快主动地将其删除。在现代 DevOps 环境中，手动将关键软件部署到服务器也同样被视为鲁莽的行为。

识别并去除不需要的可变因素。

6.5.2 让决策可逆

积极演进的系统不可避免地会意外失败。当失败发生时，开发人员需要构建新的适应度函数来防止再次失败。但是如何从失败中恢复呢？

很多 DevOps 实践可以使那些需要被撤销的决策变得可逆，例如**蓝绿部署**，该实践是 DevOps 中的常见实践，其中运维拥有两个相同的系统（可能是虚拟系统），分别标记为**蓝色**系统和**绿色**系统。如果当前的生产系统在**蓝色**系统上运行，绿色系统将作为下一次发布的预备区。当**绿色**系统准备好发布时，它将成为生产系统，而**蓝色**系统则暂时切换为后备状态。如果**绿色**系统出现错误，那么运维团队可以很顺畅地回到**蓝色**系统。如果**绿色**系统工作正常，则**蓝色**系统将成为下一次发布的预备区。

功能开关是开发人员使决策可逆的另一种常见方式。通过在功能开关控制下部署变更，开发人员可以将这些功能发布给小部分用户（金丝雀发布）来验证变更。如果某个功能发生意外，开发人员可以关闭功能开关，修复错误后重新发布。记住，确保移除那些过时的功能开关。

使用功能开关能显著降低上述情况下的风险。服务路由（基于请求上下文将请求路由到指定的服务实例）是微服务系统中用于进行金丝雀发布的另一种常见方法。

尽可能多地使决策可逆（在不过度工程化的前提下）。

6.5.3 演进优于预测

> ……据我们所知，我们已经知道一些，我们知道我们已经知道一些，我们还知道，我们有些并不知道，也就是说，我们知道有些事情我们还不知道，但是，还有一些，我们并不知道我们不知道，这些我们不知道的，我们不知道。
>
> ——美国前国防部长 Donald Rumsfeld

未知的未知问题是软件系统的大敌。很多项目始于一系列**已知的未知问题**，例如开发人员知道他们需要学习领域知识和新技术。然而，项目也会受到**未知的未知问题**的影响，例如，没人料到的问题却意外发生了。这便是为什么那些提前进行了大量设计的项目进展都很曲折的原因，架构师无法就未知的未知问题进行设计。

> 由于未知的未知问题，所有架构都将是迭代式的，敏捷实践只是较早地意识到了这一点。
>
> ——Mark Richards

虽然未知问题无法避免，但是我们知道动态平衡导致了软件开发领域的不可预见性。于是，我们倾向于将演进能力构建到软件中，如果项目可以轻松地应对变化，那么架构师就不再需要水晶球。架构并不是孤立的前期设计活动，项目在其整个生命周期里持续变化着，一些变化是明确的，另一些则不是。使用防腐层是开发人员隔离变化的常用技术。

6.5.4 构建防腐层

项目通常需要和一些提供附带管道的库耦合，例如消息队列、搜索引擎等。**抽象干扰反模式**描述了这样的场景，项目与某个外部依赖库（商业的或开源的）建立了太多连接。一旦开发人员要升级或更换该库，他们会发现调用该库的很多代码会带有基于该库的抽象假设。领域驱动设计中包含了针对这一现象的保护措施，叫作**防腐层**。示例如下。

头脑灵活的架构师在做决定时会遵循**最后责任时刻**（The Last Responsible Moment）原则，架构师以该原则避免项目中的常见隐患——过早引入复杂度。我们曾断断续续经手过某个使用 Ruby on Rails 的客户项目，该客户管理车辆的批发销售。在应用上线之后，出现了意外的业务场景。事实证明，二手车经销商在上传新的车辆图片到拍卖网站时，倾向于大批量地上传，无论是车辆数目还是每辆车的图片。我们发现，和人们不信任二手车经销商一样，经销商之间也互不信任。每辆车都必须有能涵盖车辆每个部分的照片。用户在上传开始之后能通过某种 UI 机制（例如进度条）查看进度，或者在晚些时候查看批量上传任务是否完成。用技术语言来说，他们需要异步上传功能。

消息队列是解决这种问题的传统架构方案，于是团队讨论是否将开源消息队列引入架构中。在这个时刻，很多项目都会遇到一个常见的陷阱："我们知道我们终将需要使用消息队列来处理很多事情，所以直接选择最炫酷的那个吧。"这个方法会带来**技术债**，所谓技术债就是项目中本不应该存在的部分，它会导致项目缺失理应存在的部分。很多开发人员将复杂的遗留代码视为唯一的技术债，但项目还会因为早期的复杂度而在无意中引入技术债。

对于这个项目，架构师鼓励开发人员寻求相对简单的方法。某位开发人员发现了 BackgrounDRb，这是一个非常简单的开源库，基于关系型数据库模拟了单个消息队列。架构师清楚这个简单的工具可能永远无法扩展以应对将来的某些问题，但他没有提出异议。他将该工具置于 API 之后以便于替换，而不去预测将来的使用场景。当处于"最后责任时刻"时，问自己几个问题，例如："我现在就必须做决定么？""能否在不延误工作的情况下晚点做决定？""我可以做什么来应付当前的需求并且之后能轻松加以调整？"

大概一年后，出现了第二个异步需求：围绕销售的定时事件。架构师评估了情况，决定增

加一个 BackgrounDRb 实例来满足该需求。又过了一年，第三个异步需求出现了，这次是要持续更新缓存和摘要。团队意识到当前的方案已经无法应付新的需求了。然而，他们现在已经清楚应用需要什么异步行为了。于是将项目切换到了 Starling，一个简单、传统的消息队列。由于有接口将原有方案进行了隔离，两位开发人员在一个迭代周期（一周）内完成了这个转换，并且没有影响该项目中其他开发人员的工作。

因为架构师构建了防腐层并提供了接口，所以替换某个功能就变成了机械性的操作。构建防腐层鼓励架构师思考他们想从库中获得什么**语义**，而不是特定 API 的调用**语法**。但这并不是**将一切都抽象**的理由！有些开发者社区喜欢抢先构建抽象层，这会干扰后序的工作。当你必须调用工厂方法（`Factory`）来获得某个远程接口的代理（`Proxy`）时，就明白其中的痛苦了。所幸大多数现代编程语言和 IDE 能让开发人员**即时**抽取接口。如果与项目绑定的某个过时的库需要更改，IDE 可以代表开发人员**抽取接口**，这样便创建了一个即时（JIT）防腐层。

构建即时防腐层来隔离库的更新。

控制应用中的耦合点，特别是外部资源，是架构师的关键职责之一。在需要的时候添加依赖。作为架构师，需要记住依赖在提供好处的同时，还会施加约束。确保从中获得的好处多过更新和管理依赖所带来的成本。

> 开发人员熟悉工具的好处，却忽视所要做出的权衡！
>
> ——Rich Hickey，Clojure 之父

使用防腐层有助于系统的演进性。虽然架构师无法预测未来，但至少可以降低变更的成本，以免受到太多负面影响。

6.5.5　案例分析：服务模板

微服务架构被设计为**无共享**架构——组件间尽可能地解耦，遵循限界上下文原则。服务间需要避免耦合的组件主要涉及领域对象、数据库模式以及其他会阻碍演进的耦合点。然而，为了确保一致性，开发团队经常希望统一管理技术耦合的某些方面（这遵循了“去除不必要的可变性”的建议）。例如组件监控、日志及其他诊断工具。由于微服务架构中的组件不断变化，因此这些工具对该架构至关重要。当运维人员必须管理上千个服务时，如果服务团队忘记为服务添加监控能力，可能导致灾难性后果。该服务一旦部署，就可能消失在黑洞中，因为在这些环境中，如果服务无法被监控，那就无异于隐形。但在高度解耦的环境中，团队如何保持一致呢？

服务模板是保证一致性的常见方案。其中包含一系列预配置的公共基础设施库，例如服务发现、监控、日志、度量、认证 / 授权等。在大型组织中，由共享基础设施团队管理服务模板。服务开发团队将模板作为项目脚手架，在其中编写业务行为。如果日志工具需要升级，那么共享基础设施团队能独立于服务团队而管理升级，服务团队不用知道（关心）该变更的发生。如果发生了破坏性的变更，那么部署流水线在准备环境时就会失败，并尽快向开发人员发出警告。

这个例子很好地诠释了我们所提倡的**适当耦合**。将技术架构功能复制到各个服务会造成大量已知问题。但如果能准确地找到所需的耦合度，我们便可以推进架构的演进性而不引入新问题。

> 使用服务模板仅将合适的架构部分耦合在一起，例如基础设施组件，团队可以从耦合中获益。

服务模板体现了适应性。不把技术架构作为系统的主要结构，使得我们能更容易地将变更和架构维度准确对应起来。当开发人员构建分层架构时，每一层的变更很容易，但跨层的变更会高度耦合。随着分层架构将技术架构划分在一起，它还带来了其他问题，例如领域、安全、运维问题等。通过构建部分架构来解决技术架构问题，例如服务模板[4]，开发人员可以隔离和统一对整个架构维度的变更。第 4 章讨论了如何将架构元素视为可部署单元。

6.5.6　构建可牺牲架构

在《人月神话》中，Fred Brooks 提到了在构建新的软件系统时要做好抛弃它的计划。

> 因此，在管理上，不应该询问是否该构建一个试验性的系统然后将其抛弃。因为你一定会那样做。因此，做好抛弃它的计划，因为你终将如此。
>
> ——Fred Brooks

他认为一旦团队完成了系统构建，他们就厘清了所有未知的未知问题和正确的架构决策，这些信息开始非常模糊，团队将从下一个版本中获益。在架构层面，开发人员努力预测迅速变化的需求和特征。进行概念验证是在选择架构时获取足够信息的一种方式。Martin Fowler 对“可牺牲架构”的定义是：在概念验证成功后即被抛弃的架构。例如，1995 年，eBay 始于 Perl 脚本，在 1997 年迁移到 C++，然后在 2002 年又用 Java 改写。显然，在经历了多次架构重建后，eBay 仍取得了巨大的成功。Twitter 也深谙此道。当年 Twitter 发布时，为了快速进入市场，它由 Ruby on Rails 编写。然而，随着 Twitter 蹿红，平台无法支

注 4：服务模板是微服务架构的一部分，专门用来解决微服务架构中的技术架构问题。——译者注

持水平扩展，导致频繁宕机和可用性受限。很多早期用户都对他们的故障信标非常熟悉，如图 6-8 所示。

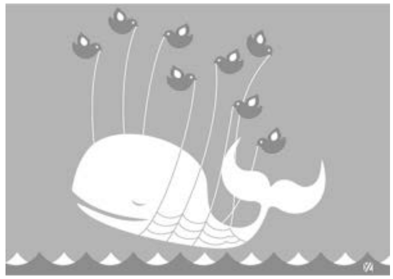


图 6-8：Twitter 出名的故障信标

因此，Twitter 重建了他们的架构，用更强大的技术重写了后端。然而，人们可以争辩说该策略是 Twitter 之所以存活的原因。因为如果 Twitter 的工程师一开始就构建最终会很强大的平台，他们进入市场的时间可能会推迟，而让 Snitter 或其他短消息服务商有足够的时间在市场上将其击败。尽管发展伴随着痛苦，但从可牺牲性架构开始，终将获得成效。

云上环境使可牺牲性架构更具吸引力，如果开发人员需要测试某个项目，在云上构建初始版本将大大降低发布软件所需要的资源。如果项目成功了，架构师可以花时间构建更合适的架构。如果开发人员对防腐层和演进式架构实践非常小心，迁移架构将会轻松些。

为了证明市场的存在，很多企业构建可牺牲性架构来实现最小可行性产品。虽然这个策略很好，但最终，团队仍会投入时间和资源来构建更强大的架构，但愿没有 Twitter 那么明显。

在 Fred Brooks 提到**第二系统综合症**时，他指出技术债会影响很多在初期很成功的项目。由于期望膨胀，小的、优雅的、成功的系统往往会演进成为塞满各种功能的庞然大物。业务人员不愿抛弃还在运行的代码，因此架构走向了一直做加法，但从不做减法的不归路。

作为某种隐喻，技术债发挥着有效的作用，因为它与项目经历共鸣，代表着设计中的缺陷，无论其背后的驱动力如何。技术债加重了项目中不当的耦合——糟糕的设计经常表现为病态耦合和其他反模式，使重建代码变得困难。在开发人员重建架构时，第一步应该清除那些以往的设计妥协，即技术债。

6.5.7　应对外部变化

外部依赖是所有开发平台的一个共同特征，其中包括工具、框架、库和其他来自互联网并（更重要的）通过互联网进行更新的资产。软件开发处在高耸的层层抽象之上，每一层抽象都建立在下层抽象之上。例如，操作系统便是开发人员无法控制的外部依赖。除非公司要编写自己的操作系统及其他所有支持代码，否则他们必须仰仗外部依赖。

经由构建工具，大多数项目会依赖于繁多的第三方组件。开发人员喜欢外部依赖，因为它们能带来好处，但是很多开发人员忽略了随之而来的代价。当我们依赖第三方代码时，开发人员必须采取防御措施来预防可能的意外，例如破坏性的变更、未经通知的删除等。管理项目的这些外部组件是构建演进式架构的关键。

破坏了整个互联网的 11 行代码

2016 年年初，JavaScript 的开发人员在某个不起眼的依赖上栽了大跟头。某个创建了很多实用小工具的开发人员，他的模块与某个商业软件重名了，而后者要求他更名，因此他有些恼火。但他并没有照做，而是删除了 250 多个模块，包括一个名为 left-pad.io 的库，它通过 11 行代码来为字符串左边添加零或空格（如果 11 行代码也称得上“库”）。不幸的是，很多主要的 JavaScript 项目（包括 Node.js）都依赖该库。在它消失后，所有人的 JavaScript 部署都无法进行。

JavaScript 的仓库管理员通过恢复代码恢复了整个体系，这是前所未有的，但它引发了社区对于如何更好地管理依赖的更明智的深层次讨论。

这个故事教给架构师两点教训。第一，铭记外部依赖在带来好处的同时还需要付出成本。我们需要确保收益大于成本。第二，不要让外部力量影响构建的稳定性。如果某个上游需要的依赖突然消失，那么应该拒绝该变更。

Edsger Dijkstra 是计算机科学领域的传奇人物，1968 年因论文“Go To Statement Considered Harmful”而出名，文中他批评了当时非结构化编程的最佳实践，最终引发了结构化编程革命。此后，“被视为有害的”成为了软件开发界的一种修辞手法。

> 传递依赖管理被视为有害的。
>
> —— Chris Ford（和 Neal 没有关系）

Chris 认为，我们只有意识到问题的严重性才能找到解决方案。有时我们无法找出问题的解决方案，但我们需要格外留意它，因为它会严重影响演进式架构。稳定性是持续交付和演进式架构的共同基础。开发人员无法基于不确定因素构建可重复的工程实践。而允许第三方修改核心依赖背离了这一原则。

我们建议开发人员采取更加积极的方法来管理依赖。以**拉取**的方式获取外部依赖是开启依赖管理的良好开端。例如拉取，设置一个内部版本控制仓库作为第三方的组件商店，然后将外部的变更视为对仓库的拉取请求。如果变更是有益的，那么将其纳入体系中。如果某个核心依赖突然消失，那么就应该将该拉取请求视为破坏稳定的因素并将其拒绝。

秉持持续交付思维，第三方组件库使用自己的部署流水线。当组件发生变更时，部署流水线合并修改，接着执行构建并对受影响的应用进行冒烟测试。如果成功，则保留变更。因此，第三方依赖使用与内部应用相同的工程实践和内部开发机制，有效地模糊了自研代码和第三方依赖之间通常不重要的区别，因为它们终将成为项目中的代码。

6.5.8 更新库与更新框架

架构师在**库**和**框架**之间做出了一般区分，简单地将其描述为“开发人员的代码会调用库，而框架会调用开发人员的代码”。通常，开发人员从框架中派生出子类（框架反过来调用这些派生出的类），这便是框架调用代码的原因。相反，库代码通常是一系列相关的类或函数，开发人员按需调用它们。由于框架调用开发人员的代码，导致了框架的高度耦合。相反，库通常更为实用，耦合度也更低，例如 XML 解析器、网络库等。

我们青睐库，因为它们引入应用的耦合更少，使得在技术架构演进时易于置换。

工程实践是我们区别对待库和框架的另一个原因。框架通常具备多种能力，例如 UI、对象关系映射、MVC 脚手架等。由于应用基于框架搭建，所以应用的所有代码都会受到框架变更的影响。很多人都真切地感受过这样的痛苦——当基础框架因两个主要版本而过时，那么升级框架需要极大的付出（伴随着巨大的痛苦）。

由于框架属于应用的基础部分，团队必须积极地将其更新。库所形成的脆弱集成点比框架更少，团队更新库时会更自由。一种非正式的管理模式将框架的更新视为推动式更新，将库的更新视为拉动式更新。当基础框架更新时（其输入 / 输出耦合数量高于某个特定阈值），只要新版本稳定，并且团队能分配出时间，那么就应该应用该更新。尽管这会花费时间和精力，但如果团队无限期地拖延该更新，最终所花费的时间将远不止这些。

由于大多数库提供实用的功能，团队可以在新的、有需要的功能出现时再花费精力更新它们，采取“按需更新”的模式。

积极地更新框架依赖，消极地更新库。

6.5.9 持续交付优于快照

很多依赖管理工具通过**快照机制**支持持续开发。构建快照最初是为了标明那些差不多准备好发布但仍在开发中的组件，它暗示着代码还可能定期更新。一旦带上了版本号，那么“快照”（-SNAPSHOT）的标记将被移除。

开发人员使用快照是因为过去大家认为测试困难且耗时，这导致开发人员尝试区分变化的内容和没变化的内容。

在演进式架构中，所有事务都在不断变化，需要通过构建工程实践和适应度函数来适应变化。例如，当项目有着出色的测试覆盖率和部署流水线时，开发人员可以通过自动的部署流水线测试每个组件的每次变更。开发人员没有理由为项目的每个部分保留某个特殊的仓库。

相比快照，我们倾向于通过持续交付来处理（外部）依赖。

快照是某个开发时期的产物，当时全面测试还不普遍，存储成本很高，验证也很困难。如今，现代工程实践可以高效处理组件依赖。

持续交付建议以更细致的方式思考依赖，我们在此重复说明。目前，开发人员只有**静态**依赖，通过在构建文件中作为元数据记录的版本号来关联。然而这对于现代项目还不够，我们需要某种机制预测更新。因此，正如书中所建议的，开发人员应该为外部依赖引入两个新定义：**流动的**依赖和**被守护的**依赖。通过部署流水线机制，流动的依赖尝试自动地将自己更新到新版本。例如，假设订单服务（`order`）流动地依赖于框架（`framework`）的 1.2 版本。当框架升级到 1.3 版本时，订单服务尝试通过部署流水线合并该更新，在项目的任意部分发生变化时该部署流水线会重建。如果部署流水线运行完成，则组件间的流动依赖也同时完成更新。然而，如果失败的测试、破坏了钻石依赖等问题阻碍了流水线的顺利完成，那么该框架的 1.2 版本的依赖将更新为**被守护的**依赖。这意味着开发人员应该找出问题并加以解决，以恢复流动的依赖。如果组件实在无法兼容，开发人员则应创建对旧版本的永久静态引用，以避免将来的自动更新。

目前流行的构建工具都未支持该级别的功能，开发人员必须基于现有的工具构建这种智能功能。然而在演进式架构中，这种依赖模型表现得相当不错，其中生产周期作为关键的基础值，和其他很多关键指标成正比。

6.5.10 服务内部版本化

在任何集成架构中，开发人员不可避免地需要随着系统行为的演进对服务端点进行版本化。版本化端点有两种常用的方式：**版本号**或**内部版本化**。对于版本号，当破坏性的变更发生时，开发人员会创建新的、通常包含版本号的端点名。这使得旧的集成点继续调用旧的服务，而新的集成点则调用新的版本。另一种替代方案是内部版本化，调用方无须修改端点，相反，开发人员在服务端构建逻辑来确定调用方的上下文，从而返回正确的版本。相比调用应用时指定版本号，使用固定名称的好处是耦合更少。

无论是哪种情况，都应该严格限制所支持的版本数量。更多的版本会增加测试和其他工程的负担。建议一次只支持两个版本，并且只是暂时支持。

在版本化服务时，我们倾向于内部版本化，一次只支持两个版本，而不是用版本号。

6.6 案例分析：PenultimateWidgets的评分服务演进

PenultimateWidgets 应用了微服务架构，因此开发人员可以进行小的变更。我们仔细看看其中某个变更的细节，切换星级评分服务，如第 3 章所述。目前，PenultimateWidgets 有一个星级评分服务，其构成如图 6-9 所示。

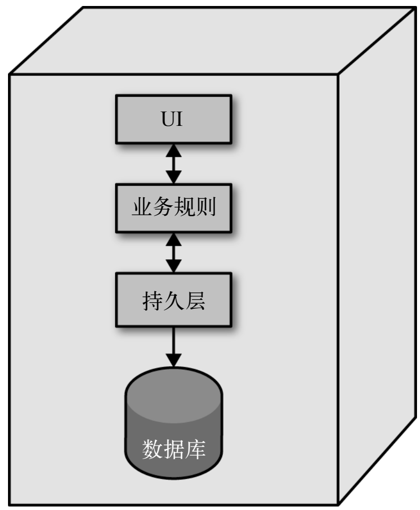


图 6-9：PenultimateWidgets 星级评分服务的内部结构

如图 6-9 所示，星级评分服务由一个数据库和分层架构组成，分层架构又包含持久层、业务规则和 UI。并非所有的 PenultimateWidgets 微服务都包含 UI。有些主要是信息服务，而其他有 UI 的服务则与服务行为紧密耦合，正如星级评分服务一样。数据库是传统的关系型数据库，通过字段来记录对特定商品的评分。

当团队决定更新该服务以支持半星评分时，他们修改了原有的服务，如图 6-10 所示。

如图 6-10 所示，团队向数据库中添加了新的字段来处理额外的数据——评分中是否包含额外的半星。架构师还向服务中添加了一个代理组件来解决服务边界的不同返回。星级评分服务没有强迫服务调用者理解服务的版本号，而是解析请求类型，并按照请求的格式返回。这是一个以**路由**为演进机制的例子。只要还有服务需要旧的星级评分服务，那么它就可以保持这个状态。

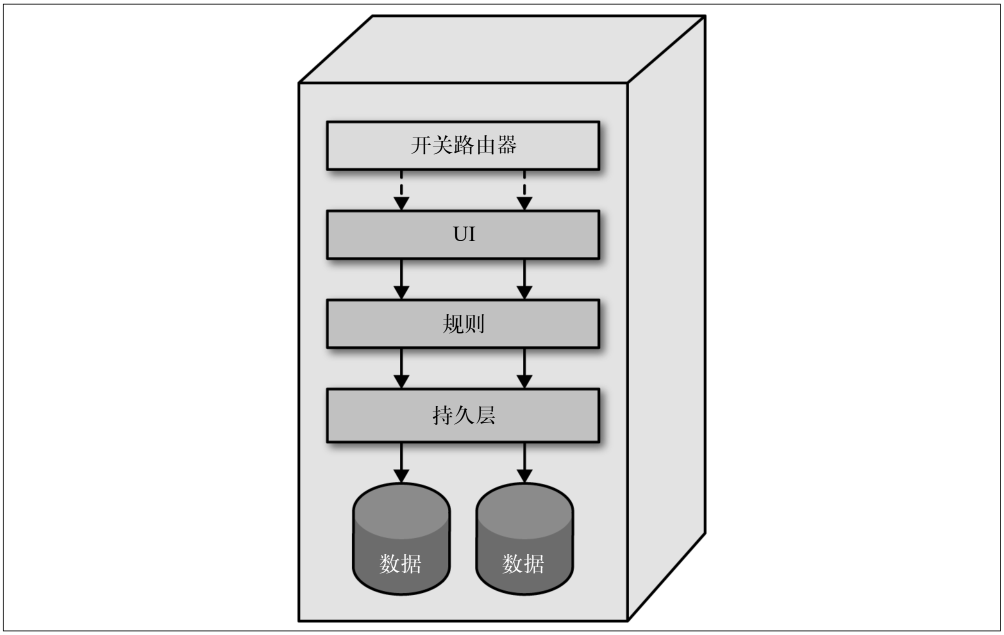


图 6-10：在过渡阶段，星级评分服务同时支持两种类型

当最后一个依赖全星评分的服务演进到新版本时，开发人员便可以移除旧的代码路径，如图 6-11 所示。

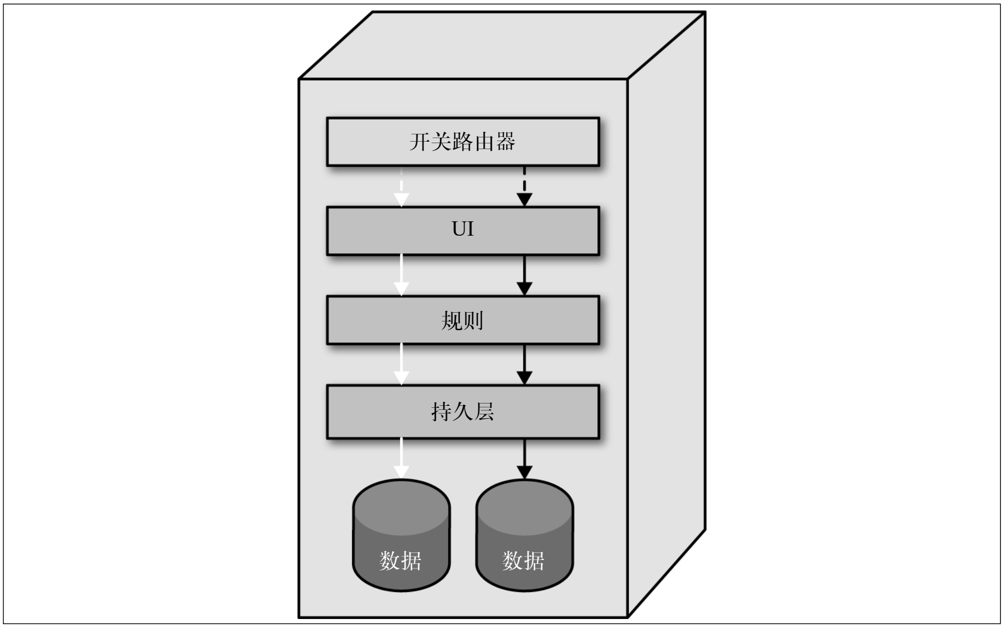


图 6-11：星级评分服务的最终状态，仅支持新的评分类型

如图 6-11 所示，开发人员可以移除旧的代码访问路径，或许还会移除处理不同版本的代理层（抑或保留它来支持将来的演进）。

在这个案例中，PenultimateWidgets 的变更从数据演进的角度来看并不难，因为开发人员可以进行添加式的变更，这意味着可以将变化添加到数据库模式中，而不是修改数据库模式。那么由于新功能导致必须修改数据库的情况又该怎么做呢？请参考第 5 章关于演进式数据的讨论。

第 7 章

演进式架构的陷阱和反模式

我们用了很多篇幅探讨架构中适当的耦合，然而现实中存在很多**妨碍**项目演进的耦合。

项目中有两种错误的工程实践——**陷阱**和**反模式**。很多开发人员使用行话“反模式”来表示“不好的模式”，但其真实含义更加微妙。软件的反模式包含两层含义。首先，反模式是一种实践，开始看起来不错，但结果证明是错的。其次，大多数反模式都有更好的替代方案。很多反模式只有在事后才被架构师注意到，因此很难避免。**陷阱**表面上像是个好主意，但很快便显露出缺点。本章将详细介绍它们。

7.1 技术架构

本章关注业内的常见实践，特别是那些妨碍团队构建演进能力的实践。

7.1.1 反模式：供应商为王

一些大型企业购买 ERP（企业资源计划）软件来处理常规业务任务，例如会计、库存管理及其他常见琐事。如果企业愿意改变其业务流程和其他决策来适应此类工具，那么这样做是可行的。如果架构师了解其中的好处和限制，可以战略性地采用这种方式。

然而，很多组织对这类软件的使用过于激进，导致了**供应商为王反模式**，一种完全围绕供应商产品构建的架构，将组织和工具病态地耦合。购买了供应商软件的公司计划通过插件扩充软件包，以丰富供应商软件的核心功能来匹配其业务。然而，很多时候无法将 ERP 定制到满足所有需求，开发人员发现他们受到了 ERP 的制约，这一方面来自于工具的限制，

另一方面来自于 ERP 是事实上的架构核心。换句话说，架构师让供应商成为了架构的王者，左右了未来的决策。

想要规避这种反模式，需要将所有软件都视为集成点，即便起初它具有广泛的职责。如果在一开始便假设集成，开发人员可以更容易地替换掉那些对其他集成点无用的行为，废除“王者”。

无论从技术还是从业务流程的角度来看，将外部工具或框架置于架构的核心会严重限制架构的演进能力。开发人员在技术上受到了供应商选择的制约，例如持久层、基础设施以及其他限制。从商业角度来看，大型封装工具最终会受到 7.1.3 节“反模式”问题的影响。从业务流程的角度来看，这种工具无法支持最佳的工作流，这也是其副作用。大多数公司最终屈服于这种框架，不再尝试定制这类工具，而是修改自己的流程。越多公司这样做，公司间的差异变得越小，当然，如果差异化不是竞争优势，这或许是可以接受的。

“让我们停止工作并宣布它的成功”，这是现实世界中处理 ERP 软件时开发人员通常会遇到的处事原则。公司不愿承认 ERP 不起作用，因为他们投入了大量时间和金钱。没有 CTO 愿意承认他们浪费了数百万美元，也没有供应商想承认数年来糟糕的实践。因此，尽管很多承诺的功能并未实现，大家仍达成共识：停止工作并宣布其成功。

> 不要将你的架构耦合，使得供应商为王。

与其沦为供应商为王反模式的受害者，我们不如将供应商产品视为集成点。开发人员可以在集成点间构建防腐层，从而避免架构受到供应商工具变更的影响。

7.1.2 陷阱：抽象泄漏

> 所有重大的抽象在某种程度上都会泄漏。
>
> —— Joel Spolsky

现代软件构建于层层抽象之上：操作系统、框架、依赖等。开发人员构建抽象来摆脱在最底层无尽的思考。如果开发人员需要将来自硬件驱动的二进制数字转换为文本来进行编程，他们将无法完成任何工作。现代软件成功的原因之一在于我们能建立有效的抽象。

但是抽象也是有代价的，因为没有抽象是完美的，如果有，那么它将不再是抽象，而是实际存在。正如 Joel Spolsky 所言：所有重大的抽象机制都会泄漏。这对开发人员来说是个问题，因为人们相信抽象总是准确的，但抽象往往会出现意外偏差。

最近，随着技术栈更加复杂，抽象干扰的问题愈发严重。图 7-1 展示了 2005 年的典型技术栈。

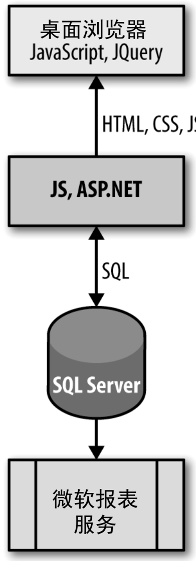


图 7-1：2005 年的典型技术栈

图 7-1 展示了 2005 年的典型软件技术栈，其中方框内的供应商名字依据实际情况而变化。随着时间推移，软件越来越专业化，技术栈变得越来越复杂，如图 7-2 所示。

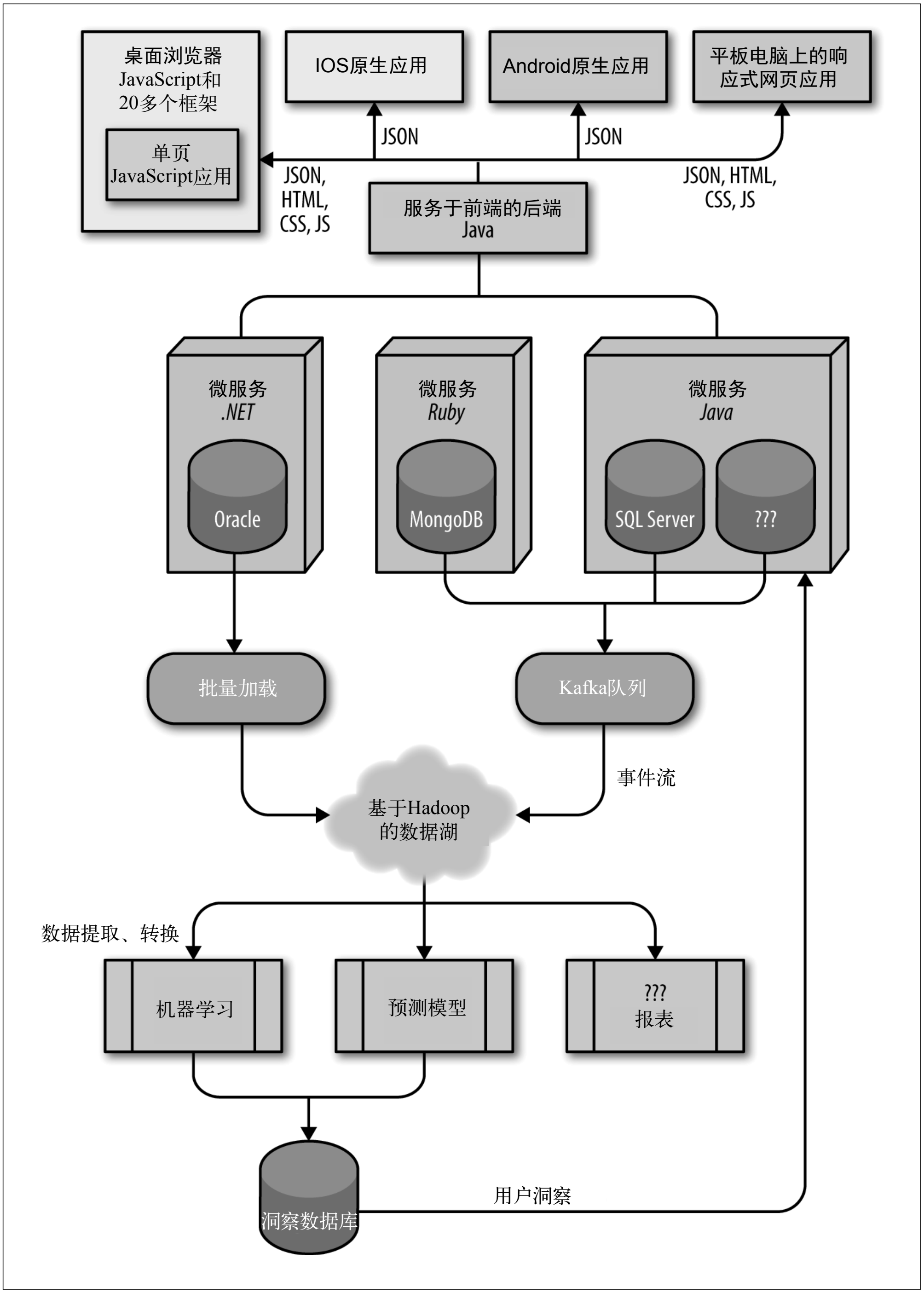


图 7-2：2016 年的典型软件技术栈，包含了很多活动的部分

如图 7-2 所示，软件体系的每个部分都已扩展并且变得更加复杂。随着开发人员所面对的问题越来越复杂，解决方案亦是如此。

底层抽象破坏会导致意外的灾难，即原始抽象泄漏，它是技术栈日渐复杂带来的副作用之一。如果最底层的某个抽象出现错误会怎样呢？例如，看似无害的数据库调用带来的意外副作用。由于层层抽象的存在，错误将蜿蜒至技术栈顶层，或许会沿着路径转移，最终在 UI 上表现为深度嵌套的错误信息。随着技术栈愈发复杂，调试和取证分析变得更加困难。

> 始终保持对当前抽象层以下至少一个抽象层的完全理解。
>
> ——许多软件专家

虽然理解下层抽象是个好建议，但它正变得越来越困难，因为软件变得越来越专业化和复杂。

技术栈复杂度的增长印证了动态平衡问题。不只是体系在变化，其组成部分随着时间推移也会变得更加复杂并交织在一起。适应度函数（保护演进式变更的机制）能够保护架构中脆弱的连接点。架构师将关键集成点上的不变量定义为适应度函数，并在部署流水线运行它们，确保抽象不会意外泄漏。

了解复杂技术栈的脆弱部分，并通过适应度函数自动保护它们。

7.1.3 反模式：最后10%的陷阱

在抽象范围的另一端存在着另一种复用陷阱，它隐藏在套装软件、平台和框架中。

Neal 曾担任某咨询公司的 CTO，使用各种第四代编程语言（4GL）为客户构建项目，其中包括微软 Access。最终在他的帮助下，客户做出了去 Access 的决定，因为他们发现每个 Access 项目开始时都异常成功但最终都失败了。Neal 非常好奇其中缘由。他和一位同事发现在 Access 和其他 4GL 盛行之时，客户 80% 的需求都能快速且轻松地实现。快速应用开发工具塑造了当时的环境，它们具有拖放式 UI 开发和其他优点。然而，客户需求的 10% 虽有实现的可能，但是极其困难。因为工具、框架或编程语言都没有提供相应功能。于是聪明的开发人员想到了一些办法，通过破解来达到目的，例如，在静态任务中添加脚本、链接方法等。但破解只能让完成度从 80% 上升到 90%。最终，这些工具仍然无法完全解决问题，我们称其为“最后 10% 的陷阱”，即所有项目都存在缺憾。虽然使用 4GL 可以方便、快速地构建简单应用，但它们无法满足现实世界的需求。开发人员因此回到了通用语言。

> **IBM 的旧金山项目**
>
> 20 世纪 90 年代后期，IBM 开始了一个野心勃勃的计划——编写业务软件的最后一部分。一组开发人员着手设计一系列可复用的业务组件，这些组件是用当时的企业级 Java 编写的，并将所有的业务功能封装到广泛的类别中，例如账目、库存、销售等。IBM 一度声称该项目是世上最庞大的 Java 项目。在项目交付了头几个核心模块后，开发人员开始使用这个框架，并导致其走向灭亡。因为很多功能都是多余的，同时又缺失了许多关键功能。
>
> 旧金山项目体现了架构师和开发人员的终极妄想，他们想凭借自己天生直觉来将一切事物分门别类。但整洁的解决方案无法解决现实世界中一些混乱的事物，例如业务流程。
>
> 旧金山项目以失败告终，因为人们逐渐认识到了一个令人深省的事实，即无论开发人员多么努力，他们都无法将事物提炼得足够精细，这便是**无限回归**问题的一部分：一些命题依赖于其他命题而成立，没有止境。在软件领域中，无限回归表现为人们想要用终级的细节详细描述任何事物，但在任何现有细节之下总是存在另一层更细粒度的细节。

7.1.4 反模式：代码复用和滥用

在软件行业中，我们从他人构建的可复用框架和库中受益匪浅，它们通常是开源软件，可以免费使用。复用代码显然很好，然而，任何美好事物都不能被滥用，很多公司因滥用代码给自己造成了麻烦。每个企业都希望复用代码，因为软件看起来模块分明，像电子元件一样。然而，尽管在真正模块化的软件中的确如此，但它却难以实现。

> 复用软件更像是器官移植而不是拼装乐高积木。
>
> ——John D. Cook

虽然一直以来开发语言的设计者向开发人员承诺的是“乐高积木”，但似乎我们还是只有“器官”。复用软件很难并且不会自动出现。很多管理者乐观地认为开发者编写的任何代码都可复用，但事实并非总是如此。很多公司尝试并成功编写出真正可复用的代码，但这是有意为之并且困难重重。开发人员通常花费大量时间尝试构建可复用的模块，结果却几乎无法复用。

在 SOA 中，常见的实践是找到公共的部分并尽可能地复用。例如，设想某个公司有两个上下文：`Checkout` 和 `Shipping`。在 SOA 中，架构师发现两个上下文中都存在 `Customer` 的概念。因而鼓励他们将两个 `Customer` 整合到一个 `Customer` 服务中，将 `Checkout` 和 `Shipping` 耦合到这个共享的服务中。架构师努力实现 SOA 中的终极**规范**——所有概念都只有一个（共享的）归属。

讽刺的是，开发人员为了代码复用所付出的努力往往适得其反。为了复用代码，需要引入额外的选项和决策点以适应不同的用途。开发人员为实现可复用所添加的钩子越多，对代码的基本**可用性**损害越大。

> 代码复用性越高，其可用性越低。

换句话说，代码的易用性和复用性往往成反比。当开发人员构建可复用的代码时，他们必然会为了将来开发人员以各种方式使用该代码添加特性。所有针对未来的特性都使得开发人员更难将代码用于单一目的。

微服务避免代码复用，遵循**重复优于耦合**的理念。该理念认为复用意味着耦合，因此微服务架构是极度解耦的。然而，微服务的目标并不是追求重复，而是隔离领域内的实体。那些共享通用类的服务不再独立。在微服务架构中，`Checkout` 和 `Shipping` 应该各自拥有其 `Customer` 的内部呈现。如果它们需要就客户信息进行协作，它们会把相关信息发送给对方。架构师不会在架构中调合不同版本的 `Customer`。复用所带来的好处是虚幻的，除了其自身缺陷，它还会引入耦合。因此，虽然架构师了解重复的缺点，但他们利用重复抵消了耦合过多对架构的局部损害。

复用代码可以是资产，也可能是潜在的责任。我们要确保代码中引入的耦合点不会和其他架构目标产生冲突。例如，微服务架构通常使用服务模板（6.6.5 节介绍过）来将服务的各个部分耦合在一起，帮助统一特定的架构关注点，例如监控和日志。

7.1.5　案例研究：PenultimateWidgets中的复用

对于 PenultimateWidgets 的管理功能来说，其专用网格对数据输入的需求高度明确。由于应用需要在多个位置使用该视图，PenultimateWidgets 决定构建一个可复用的组件，包括 UI、校验和其他有用的默认行为。使用该组件，开发人员可以轻松构建新的、丰富的管理界面。

然而，所有架构决策都必然会伴随着一些折中。随着时间推移，组件团队成了组织中的孤岛，占用了 PenultimateWidgets 最好的几名开发人员。使用该组件的团队需要向组件团队请求新功能，其中充斥着错误修复和功能请求。更糟糕的是，底层代码并没有跟上现代网页标准，导致实现新功能变得非常困难甚至不可能。

虽然 PenultimateWidgets 做到了复用，但最终导致了瓶颈效应。复用的一个好处是开发人员可以快速构建新的软件。但是，除非组件团队可以跟上动态平衡的创新步伐，否则技术架构组件的复用最终注定会沦为反模式。

我们并非建议团队避免构建可复用的资产，而是建议不断对其进行评估以保证它们仍能提供价值。在 PenultimateWidgets 的案例中，当架构师意识到组件成为了瓶颈时，他们打破了耦合点。任何团队都可以分叉（fork）组件代码并添加自己的新功能（只要应用团队提供支持），任何想采用新方法的团队都完全不会受到旧代码的束缚。

从 PenultimateWidgets 的经历得出以下两条建议。

> 当耦合点妨碍了演进或其他重要的架构特征时，通过分叉或重复来打破耦合点。

在 PenultimateWidgets 案例中，通过赋予团队对共享代码的所有权，他们打破了耦合。虽然会增加团队的负担，但它释放了团队交付新功能的能力。在其他情况下，或许某些共享代码可以从较大的部分中抽取出来，使得耦合更有选择性、更加平缓。

> 架构师必须持续评估架构特征的适应度，保证它们仍在提供价值，避免沦为反模式。

架构师在当时做出的正确决定，随着时间推移，由于动态平衡等因素的变化，往往会变得不再正确。例如，架构师将系统设计为桌面应用，但随着用户习惯的改变，业界将其引向了网页应用。最初的决定并没有错，但环境意外地改变了。

7.1.6　陷阱：简历驱动开发

架构师迷恋软件开发领域的新发展，并迫不及待地想要尝试。然而，要选择出高效的架构，他们必须仔细了解对应的问题域并选择最合适的架构，这样才能提供最理想的能力并且破坏性约束最小。当然，除非架构师陷入了**简历驱动开发**的陷阱——为了用这些知识丰富自己的简历而选择框架和库。

> 不要为了架构而构建架构，构建架构是为了解决问题。

在选择架构前，要始终理解问题域，不要本末倒置。

7.2 增量变更

软件开发中很多因素会妨碍增量变更。几十年来，编写软件的目标没有考虑敏捷性，而是围绕着降低成本、共享资源和其他一些外部约束。因此，很多组织缺乏能够支持演进式架构的基础。

正如《持续交付》一书所提到的，很多现代工程实践都支持演进式架构。

7.2.1 反模式：管理不当

软件架构并非处于真空之中，它通常反映了设计时所处的环境。十年前，操作系统是非常昂贵的商品。数据库服务器、应用服务器和运行应用的整个基础设施也是如此。为了应对这些现实世界的压力，架构师通过设计架构来最大程度地共享资源。当时涌现了很多架构模式，例如 SOA。在那样的环境中，一种常见的管理模式应运而生，它将最大化共享资源作为节省成本的措施。很多工具的商业动机都是从这种趋势中发展而来，例如应用服务器。然而从开发的角度来看，由于无意的耦合，在同一台主机上打包多个资源并不可取。无论共享资源间隔离得多么好，资源竞争终将出现。

在过去的十年中，软件开发领域的动态平衡发生了很多变化。如今，开发人员可以构建组件高度隔离的架构（例如微服务），来消除因共享环境加剧的意外耦合。但是很多公司依然坚持着陈旧的管理手段。这类管理模式重视共享资源和同质化的环境。近来由于 DevOps 等运动的改进，使得这种管理模式不再适用。

> 如今，每个公司都是软件公司。
>
> ——《福布斯》杂志，2011 年 11 月 30 日

《福布斯》的这一金句想表达的是：如果一家航空公司的 iPad 版应用很糟糕，那么它最终将影响该公司的生存底线。软件能力是每个前沿公司的必备能力，对于想要保持竞争力的公司更是如此。其中便包括如何管理研发资产，例如软件运行环境。

当开发人员能够不费成本（金钱或时间）地创建虚拟机和容器资源时，看重单一解决方案的管理模式就变得不适用了。微服务领域出现了一种更好的方式。微服务架构的一个常见特征就是支持异构的环境，各个服务团队可以选择适合的技术栈来实现他们的服务，而不用按照企业标准进行统一。这与传统方式截然相反，因此当传统企业的架构师听到这个建议时会退缩。然而，大多数微服务项目的目标并不是武断地选择不同的技术，而是根据具体问题选择适当的技术。

在现代环境中，将不同技术栈统一成单一技术栈是不当的管理。这会无意将问题过度复杂化，管理决策使得实现解决方案所需的工作毫无意义地成倍增加。例如，以某个供应商的关系型数据库为标准是大型企业的常见实践，原因显而易见：项目一致性、员工的可替代

性等。然而，其给大多数项目带来了过度设计的副作用。当开发人员构建单体架构时，管理决策将影响所有项目。因此，在选择数据库时，架构师必须了解每个项目需要的能力，并考虑最复杂的情况。然而很多项目并不具有最复杂的情况。某个只有简单持久化需求的小项目，为了一致性，需要容纳整个工业级数据库的复杂度。

在微服务架构中，由于服务间不存在技术架构或数据架构的耦合，不同的团队可以选择正确的复杂度来实现其服务。其终极目标是化繁为简，保持技术栈复杂度和技术需求的一致。当团队全权负责其服务（包括运维）时，这样的划分往往效果最佳。

强制解耦

微服务架构的目标之一是技术架构的极限解耦，使得更换服务不会产生任何副作用。然而，如果开发人员都共享相同的代码库或平台，那么无耦合则需要一定程度的开发者守则和保护措施来确保耦合不会意外发生（因为复用现有代码的吸引力极强）。通过不同技术栈来构建服务是实现技术架构解耦的一种途径。很多公司都设法避免这种方式，因为他们害怕这样会导致其员工不可替代。然而，Wunderlist 的架构师 Chad Fowler 采取了相反的方式，他坚持团队应使用不同的技术栈来避免意外的耦合。他的理念是，相比开发人员的可替代性，意外耦合的危害更大。

很多公司将不同的功能封装到 PAAS 平台中，供内部使用。此举将技术选型（及耦合的可能）隐藏在了定义好的接口之后。

从大型组织的实用性管理的角度来看，我们发现**金发姑娘**[5] 管理模式效果不错：选择简单、中等和复杂三种技术栈作为标准，然后允许单个服务需求驱动技术栈的需求。这样就赋予了团队选择合适技术栈的灵活性，还能继续为企业保留标准化带来的好处

7.2.2 案例研究：PenultimateWidgets的“金发姑娘”管理

多年来，PenultimateWidgets 的架构师尝试将 Java 和 Oracle 作为标准技术栈。然而，随着构建出粒度更细的服务，他们意识到该技术栈使得小型服务复杂化了。但他们并不想完全采用“每个项目选择自己的技术栈”的微服务方式，因为他们希望一些知识和技术在不同项目间可移植。最终他们采取了“金发姑娘”管理模式，并制定了以下三种技术栈。

- 小型项目

  对扩展性和性能没有严格要求的简单项目，他们选择了 Ruby on Rails 和 MySQL 数据库。

- 中型项目

  针对中型项目，他们选择了 Go 语言，然后根据数据需求，从 Cassandra、MongoDB 和 MySQL 中选择一个作为数据库。

注 5：隐喻来自《金发姑娘和三只小熊》。——译者注

- ❑ 大型项目

针对大型项目，他们还是选择了 Java 和 Oracle，因为它们能很好地应对各种架构问题。

7.2.3 陷阱：发布过慢

持续交付中的工程实践排除了拖慢软件发布速度的因素，这些实践应该作为演进式架构成功的前提。虽然持续交付的终极目标，持续部署，对于演进式架构来说不是必需的，但软件的演进能力与其发布能力息息相关。

如果企业围绕持续部署打造工程文化，期望所有变更在通过了部署流水线设置的挑战后便进入生产环境，那么开发人员就会习惯于持续变更。另一方面，如果发布是一个需要很多专业化工作的正式流程，那么利用演进式架构的机会就会减少。

持续交付追求数据驱动的结果，从指标数据中学习如何优化项目。开发人员必须衡量事物从而了解如何优化。**生产周期**是持续交付的一个关键指标，和**交付周期**相关。交付周期是指从一个想法开始到它在软件中实现所耗费的时间。然而，交付周期中包含很多主观活动，例如估算、排列优先级等，使其成为了一个糟糕的工程指标。因此持续交付跟踪**生产周期**，即启动和完成单位工作所用的时间，这里指软件开发。生产周期从开发人员着手开发某个新功能起开始计时，当该功能在生产环境中运行时停止计时。其目标是衡量工程效率，持续交付的关键目标之一便是缩短生产周期。

生产周期对于演进式架构也至关重要。在生物学中，果蝇常用于验证遗传特征，因为他们的生命周期短，新一代出现的速度足够让生物学家观察到明确的结果。这在演进式架构中也同样成立——更快的生产周期意味着架构可以更快地演进。因此，一个项目的生产周期决定了架构的演进速度。换句话说，演进速度和生产周期成正比。表达式如下：

$$v \propto c$$

在上面的公式中，v 代表变更速度，c 代表生产周期。开发人员无法在生产周期内完成系统的演进。换句话说，团队发布软件的速度越快，那么他们便能够越快地演进系统的各个部分。

因此，在演进式架构项目中，生产周期是重要指标，更快的发布速度意味着更快的演进能力。事实上，生产周期是基于过程的原子适应度函数的理想选择。例如，开发人员构建了具备自动化部署流水线的项目，其生产周期为 3 个小时。随着时间推移，由于开发人员向部署流水线添加了更多校验和集成点，生产周期逐渐延长。由于时间是该项目的重要指标，他们设置了适应度函数，当周期时间超过 4 个小时便发出警告。一旦达到阈值，开发人员可以决定调整部署流水线的工作方式，或者决定是否可以接受 4 小时的生产周期。适应度函数适用于开发人员想监控项目的任何行为，包括项目指标。将项目关注点统一成适应度函数使得开发人员可以设置未来决策点，即**最后责任时刻**，以重新评估决策。在前面

的例子中，开发人员必须在当时决定哪一个更重要，是 3 小时的生产者周期，还是他们建立的测试。在大多数项目中，开发人员并不会注意到生产周期逐渐延长，因此也不会权衡冲突的目标，结果含糊地做出这些决定。借助适应度函数，他们可以围绕预期的未来决策点设置阈值。

> 演进的速度和生产周期成正比，生产周期越短，演进越快。

良好的工程、部署和发布实践是使演进式架构获得成功的关键，反过来又通过假设驱动开发为业务提供新能力。

7.3 业务问题

最后，我们讨论由业务问题导致的不当耦合。大多数时候，业务人员并非有意给开发人员制造麻烦，但他们的优先权会产生不当的架构决策，在无意间限制了将来的选择。下面讨论一些业务陷阱和反模式。

7.3.1 陷阱：产品定制

销售人员需要卖点。在销售人员的讽刺画像中，他们兜售产品中还不存在的功能。可见，销售人员希望软件可以无限定制。然而，这种能力需要一系列实施技术为支撑。

- *为每个客户定制*

  在这个场景中，销售人员在紧迫的时间内承诺实现特定版本的功能，迫使开发人员使用版本控制分支或标签技术来跟踪版本。

- *永久的功能开关*

  第 3 章介绍过功能开关，有时将它战略性地用于构建永久的定制功能。开发人员可以使用功能开关来为不同客户创建不同的版本，或创建“免费”版产品，让用户付费解锁高级功能。

- *产品驱动定制化*

  有些产品甚至可以通过 UI 来完成定制。在这种情况下，定制功能是应用的永久部分，需要和其他所有产品功能受到同样的维护。

功能开关和定制化的存在导致产品具有很多可能的路径排列，显著加重了测试负担。除了测试场景，为了保护可能的排列，开发人员可能需要构建更多的适应度函数。

定制也会妨碍演进能力，但我们并不是劝阻企业构建可定制的软件，企业应该实事求是地评估相关成本。

7.3.2 反模式：报表

大多数应用根据不同的业务功能有着不同的用途。例如，有些用户需要订单条目，而其他用户需要报表用于分析。组织力求覆盖业务所需的方方面面（无论是订单条目还是月度报表），特别是当这些需求来自相同的单体架构或数据库的时候。在 SOA 盛行的年代，架构师努力尝试通过相同的“可复用”服务支持所有的业务问题。他们发现服务越通用，就越需要开发人员定制使其发挥作用。

报表是单体架构中意外耦合的好例子。架构师和 DBA 想使用相同的数据库模式来支持记录和报表系统，但问题是，这种设计既无法优化记录系统，也无法优化报表系统。在分层架构中，开发人员和报表设计人员会合谋创建一种常见的陷阱，它体现了不同业务问题之间的紧张关系。架构师构建分层架构来减少意外耦合，创建隔离层来分离关注点。然而，报表并不需要单独的层支持其功能，它只需要数据。另外，在不同层间路由请求还会使延时增加。因此，很多有着良好分层架构的公司允许报表设计人员将报表和数据库模式直接耦合起来，使得在不影响报表的情况下无法变更数据库模式。这个例子很好地展示了冲突的业务目标是如何破坏架构师的工作，并使演进变得极其困难的。虽然没有人开始就打算让系统难以演进，但这是决策的累积效应。

很多微服务架构通过分离行为来解决报表问题，而微服务的隔离有利于分离但不利于整合。通常构建这类架构时，架构师使用事件流或消息队列来向领域“记录系统”数据库填充数据，每个记录系统嵌在服务架构量子中，使用最终一致性而不是事务行为。一些报表服务也会监听事件流，向针对报表优化过的非规范化数据库中填充数据。从架构的角度来看，协调也是一种耦合，使用最终一致性能让架构师免于协调，为应用程序的不同用途做出不同的抽象。

例如，在 PenultimateWidgets 的微服务架构中，领域分散在限界上下文中，每个领域拥有自己的“记录系统”。PenultimateWidgets 的开发人员使用最终一致性和消息队列来填充和通信，他们还有一些和领域服务分离的报表服务，如图 7-3 所示。

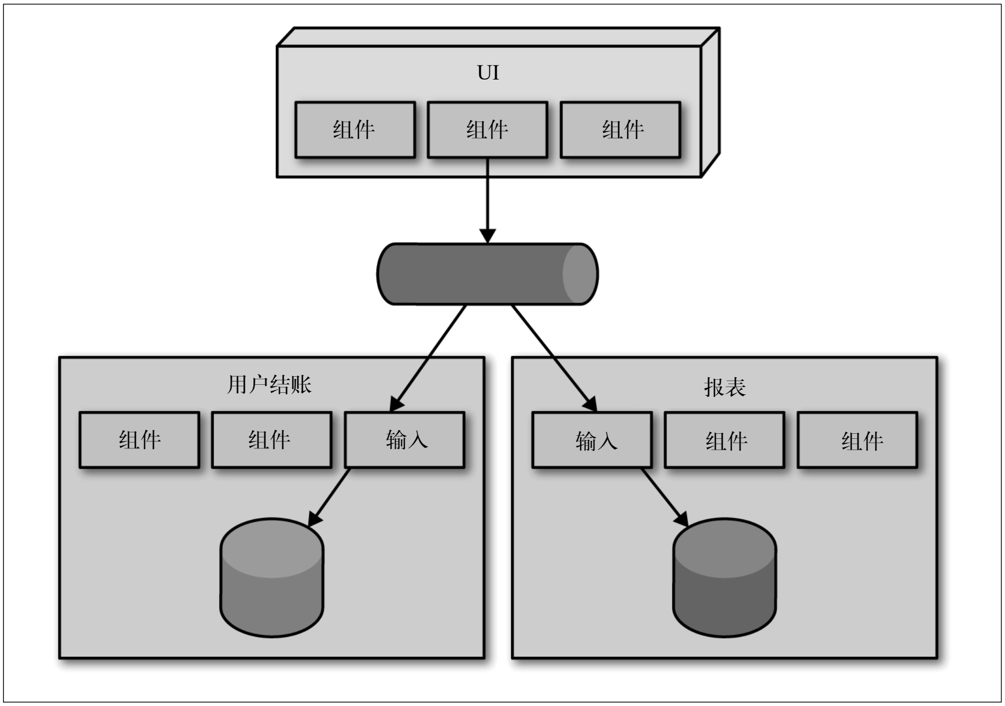


图 7-3：PenultimateWidgets 分离的领域服务和报表服务，并通过消息队列进行协调

在图 7-3 中，当界面报告了创建、读取、更新、定义（CRUD）操作时，领域服务和报表服务都会监听通知并采取相应行动。因此，这些报表服务可以在不影响领域服务的情况下处理报表问题。消除因混合领域和报表引起的不当耦合，使得每个团队可以专注于更加具体且简单的任务。

7.3.3 陷阱：规划视野

预算和规划流程通常决定了对假设和早期决策（假设的基础）的需求。然而，不去重新审视计划，规划范围越大意味着更多决策（或假设）是基于最少量的信息制定的。在早期规划阶段，开发人员花费了巨大精力于调研（通常以阅读的形式）来验证其假设。在开发人员为最终用户编写任何代码或发布软件之前，他们所学到的“最佳实践”或“同类最佳”构成了基础假设的一部分。投入到假设中的努力越多，即便在六个月内证明它们是错误的，仍会导致对其强烈的依赖。沉没成本的误区描述了受情绪投入影响的决策。简而言之，人们对某件事情投入的时间和精力越多，就越难放弃它。在软件中，它表现为**不合理的工件附件**。例如，人们在规划和文档上投入的时间和精力越多，就越可能保护其中的内容，即便有证据表明它们不准确或过时了。

理性对待软件中的定制部分。

谨防长期规划，因为它会迫使架构师做出的决策不可逆转，同时找到方法来保证多个可选方案。将大型项目分解成更小的早期可交付物，以测试架构选型和开发基础设施的可行性。在通过最终用户反馈验证所用技术确实适用他们试图解决的问题之前，架构师应该避免在实际构建软件之前采用需要大量前期投入的技术，例如大型许可和支持合同。

第8章

实践演进式架构

本章介绍实现演进式架构相关想法所需的工作。这涉及技术和业务问题，以及组织和团队的影响。我们还将对从何开始以及如何在你的公司推行演进式架构给出建议。

8.1 组织因素

软件架构广泛地影响着看似与软件无关的各种因素，包括团队影响、预算等许多方面。

在构建演进式架构时，围绕领域而不是技术能力组建团队具有许多优势和一些共同特征。

8.1.1 全功能团队

以领域为中心的团队应该是**全功能**的，这意味着每个项目角色都由该项目组成员承担。以领域为中心的团队，其目标是消除运营摩擦。换句话说，团队拥有负责设计、实现和部署其服务的所有角色，其中还包括传统上单独的角色，例如运维。但是这些角色必须改变以适应新的结构，这些角色如下所示。

- 业务分析师

  业务分析师必须与其他服务协调该服务的目标，包括其他服务团队。

- 架构师

  架构师设计架构来消除不当耦合，以简化增量变更。请注意，这里不需要像微服务那样独特的架构。一个精心设计的模块化单体应用也能以相同的能力适应增量变更（虽然架构师必须明确设计应用程序来支持这种程度的变更）。

- 测试人员

  测试人员必须习惯于跨领域集成测试带来的挑战，例如构建集成环境、创建和维护契约等。

- 运维人员

  对于 IT 结构相对传统的组织而言，划分服务并分别进行部署（通常与现有服务一同持续部署）是一项艰巨的挑战。保守派架构师天真地认为组件和运维模块化是一回事，但事实通常并非如此。自动化的 DevOps 任务是成功的关键，例如自动化的服务器配置和部署。

- 数据人员

  DBA 必须应对新的数据粒度、事务和记录系统问题。

全功能团队的目标之一便是消除协调摩擦。传统的团队彼此独立，开发人员通常需要等 DBA 做出变更或等运维人员提供资源。同一个团队包含各种角色能消除不同团队协调所产生的偶然摩擦。

为项目的每个角色配备合格的工程师非常奢侈，大多数公司都难以做到。关键的技术领域总会受到外部力量的限制，比如市场需求。因此，由于资源受限，很多公司无法如愿地组建全功能团队。在这种情况下，项目间可以尝试共享受限的资源。例如，与其为每个服务配备一名运维工程师，或许他们可以在几个不同的团队间轮转。

通过围绕领域组建架构和团队，现在可以由同一个团队处理常见的变更单元，从而减少了团队间的摩擦。以领域为中心的架构仍然使用分层架构来发挥其优势，例如关注点分离。举个例子，某个微服务的实现或许依赖于分层架构的框架，使得团队可以轻松替换某个技术层。微服务将技术架构封装在领域范围内，颠覆了传统的关系。

通过自动化 DevOps 寻找新资源

Neal 曾为一家托管服务公司提供咨询服务。该公司有十几个开发团队，各自负责明确定义的模块。然而，有一个运营团队负责管理所有的维护、环境配置、监控和其他常规任务。其主管经常收到开发人员的抱怨，他们希望在数据库、网页服务器等所需的资源上能够更快地周转。为了缓解一部分压力，他开始每周为每个团队分派一名运维人员。在那天，开发人员会非常开心，因为他们不用等待资源。遗憾的是，经理并没有足够的资源经常这么做。

至少他是这么认为的。我们发现运维团队进行的很多手动工作都是偶发且复杂的，例如，错误配置的服务器、生产商和品牌对应混乱以及其他可修复的错误。一旦所有东西都分门别类，我们便使用 Puppet 帮助他们自动化了新服务器的配置工作。此后，运维团队既有足够的成员长期参与每个项目，也有足够的人手管理基础设施的自动化。

他们既没有雇用新的工程师，也没有明显改变角色分工。相反，他们应用了现代工程实践来定期自动完成那些不应依赖人工处理的工作，从而解放了运维团队，让他们可以更好地支持开发团队。

8.1.2 围绕业务能力组织团队

围绕领域组织团队即意味着围绕业务能力组织团队。很多组织期望他们的技术架构能代表自身的复杂抽象，并与业务松散相关。这是因为传统上，架构师仅关注技术架构，通常是按照功能进行隔离的。例如，分层架构的设计初衷是为了便于替换技术架构层，而不是为了更轻松地在领域实体上工作，例如 `Customer`。这些重点大多数是由外部因素驱动的，例如过去十年中，基于成本考虑，多种架构都着重于最大化共享资源。

在大多数组织中，通过采用开源软件，架构师渐渐摆脱了商业软件的束缚。共享资源架构存在固有的问题，它会引起系统各部分间无意的干扰。现在开发人员可以创建定制的环境和功能，更容易将重点从技术架构上转移到以领域为中心的架构，进而更好地匹配大多数软件项目中的常见变更单元。

按照业务能力而非职能来组织团队。

8.1.3 产品高于项目

围绕**产品**而非**项目**来组织团队工作，是很多公司切换团队工作重点的一种机制。在大多数组织中，软件项目的工作流程是通用的。确定一个问题，组建开发团队，然后着手解决问题，直至“完成”，紧接着将软件移交给运维团队进行后续的管理、升级和维护工作。随后项目团队转向下一个问题。

这导致了许多常见问题。首先，由于团队转向了其他问题，通常很难进行漏洞修复和其他维护工作。其次，由于开发人员不参与代码运维的相关工作，因此他们不太关心质量等问题。通常，开发人员和他们运行的代码的间接层越多，他们与代码之间的联系就越少。有时这会导致在不同团队间产生对立心态，这并不奇怪，因为很多组织结构催生了员工间的冲突。

通过将软件视为**产品**，公司能在三个方面实现转变。第一，与项目的生命周期不同，产品的生命更长久。全功能团队（通常基于**康威逆定律**）与产品保持联系。第二，每个产品都有一个负责人，他会主张在体系中使用该产品，并管理其需求。第三，由于是全功能团队，团队拥有产品所需的各种角色，例如业务分析师、开发人员、质量保障人员、DBA、运维人员等。

从项目心态转变为产品心态的真正目标是得到公司的长期支持。产品团队负责产品的长期质量。因此，开发人员负责质量指标，并且更加关注缺陷。该视角还有助于为团队规划长期愿景。

> **亚马逊的“两个比萨”团队**
>
> 亚马逊以其产品团队的组织方式而闻名，他们称之为“两个比萨团队”。其理论是，两个大的比萨就够任何一个团队吃了。这种划分方式背后的动机更多是为了沟通方便而不是控制团队大小，因为在更大的团队中，成员必须和更多的人沟通。每个团队都是全功能团队，并且都奉行着“谁构建，谁运行”的理念，这意味着每个团队全权负责其服务，包括运维工作。

小的全功能团队还充分利用了人性。亚马逊的“两个比萨团队”模仿了灵长类动物群体的行为。大部分体育团队的人数都在 10 左右，并且人类学家认为前面所提到的狩猎小队的规模也与之类似。

根据人们与生俱来的社会行为构建高度负责的团队，能使团队成员更加负责。例如，设想在传统项目的结构下，某个开发人员在两年前编写的一段代码在夜里突然崩溃，迫使运维团队的工程师在当晚需要做出回应并修复问题。第二天早上，粗心的开发人员甚至可能没有意识到他无意间在半夜造成了恐慌。而在全功能团队中，如果开发人员编写的代码在夜里崩溃，那么团队成员必须做出响应，第二天早上，倒霉的开发人员无法逃避受到意外影响的队友不悦且疲惫的目光。这应该会让犯错的开发人员想成为更好的队友。

构建全功能团队可以防止不同团队间的相互指责，并让团队产生主人翁意识，激励团队成员做到最好。

8.1.4　应对外部变化

我们提倡在技术架构、团队结构等各方面都构建高度解耦的组件，从而将演进能力最大化，但在现实世界中，为了能协同解决领域问题，组件必须交互来共享信息。那么我们如何构建可以自由演进的组件，同时还维持集成点的完整性呢？

我们构建适应度函数来保护架构中的维度免于演进副作用的影响。微服务架构中的一个常见实践便是采用消费者驱动的契约，即原子集成架构适应度函数，如图 8-1 所示。

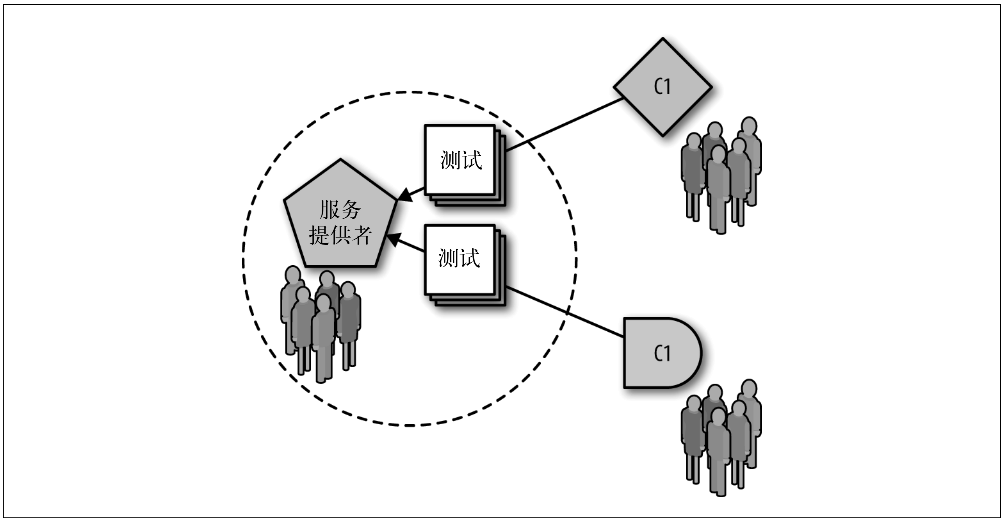


图 8-1：消费者驱动的契约，通过测试来建立服务提供者和消费者之间的契约

在图 8-1 中，**服务提供者**向其消费者 C1 和 C2 提供信息（通常以轻量的格式）。在消费者驱动的契约中，信息消费者构建一组测试来封装他们对服务提供者的需求，并将这些测试移交给服务提供者，服务提供者承诺始终让这些测试通过。因为这些测试覆盖了服务消费者所需的信息，所以服务提供者能以不破坏这些适应度函数的方式演进。在图 8-1 所示的场景中，**服务提供者**除了运行自己的测试组件外，还代表所有消费者运行这些契约测试。我们将采用这类适应度函数简单称为**工程安全网**。当可以轻松构建适应度函数来维护集成协议的一致性时，就不应该手动处理这类杂事。

演进式架构的增量变更默认开发团队具备一定的工程成熟度。例如，如果团队正在采用消费者驱动的契约，但他们的构建偶尔会损坏几天，那么他们无法得知集成点是否仍然有效。采用工程实践通过适应度函数来监督实践可以为开发人员减轻做大量手动工作的痛苦，但这需要一定的成熟度才能成功。

8.1.5 团队成员间的连接数

很多公司意识到，大型开发团队的效果不佳，J. Richard Hackman（知名团队动力专家）解释了该现象的原因。这里需要维护的并不是人数，而是人与人之间的连接数。他采用下面的公式来确定人与人之间的连接数，其中 n 表示人数。

公式 8-1 人与人之间的连接数

$$\frac{n(n-1)}{2}$$

在公式 8-1 中，随着人数的增长，连接数激增，如图 8-2 所示。

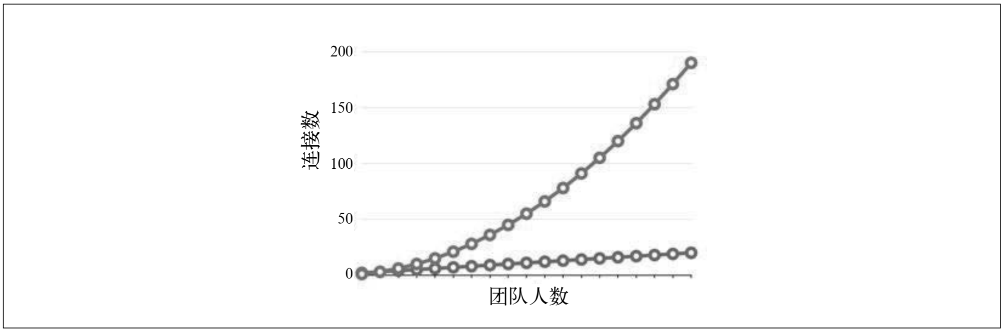


图 8-2：随着人数增长，连接数激增

在图 8-2 中，当团队人数达到 20 时，他们必须管理 190 个连接；当组员增长到 50 个时，连接数则增长为惊人的 1225。因此，构建小型团队是为了减少沟通连接。并且，为了消除不同团队间协作所产生的人为摩擦，这些小型团队需要是全功能团队。

每个团队无须了解其他团队所做的事情，除非团队之间存在集成点。即便如此，也应该使用适应度函数保证集成点的完整性。

尽量减少开发团队之间的连接数。

8.2 团队的耦合特征

企业的组织和管理其结构的方式显著影响着软件的构建和架构方式。下面探索不同的组织和团队在构建演进式架构时的难易程度。大多数架构师不会考虑团队结构对架构耦合特征的影响，但其影响不容忽视。

8.2.1 文化

> **文化**（名词）：特定人群和社会的想法、习俗及社会行为。
>
> ——《牛津词典》

架构师应该关注工程师构建系统的方式并注意组织所奖励的行为。架构师选择工具和进行设计的活动和决策过程会影响软件的演进能力。好的架构师会担任领导角色，为开发人员构建系统确立技术文化和设计方法。他们教授和支持工程师构建演进式架构所需的技能。

架构师能通过提下列问题了解团队的工程文化。

- 是否团队所有人都知道什么是适应度函数，并考虑了新工具或产品选型对演进新适应度函数的影响？
- 团队是否衡量了系统与所定义的适应度函数的匹配程度？
- 工程师是否理解内聚和耦合？
- 是否讨论了什么领域和技术概念该整合到一起？
- 团队是基于变更能力还是基于他们想学习的技术来选择解决方案的？
- 团队对业务变更如何做出反应？他们是否难于完成小的变更，或在小的业务变更上花费了太多时间？

> 告诉我你的衡量标准，我就告诉你我会如何行动。
>
> ——Eliyahu M. Goldratt 博士，*The Haystack Syndrome*

如果团队不习惯改变，那么架构师可以引入实践来优先应对这一点。例如，当团队考虑采用某个新的库或框架时，架构师可以让团队通过快速试验来进行明确的评估，看看新的库或框架会引入多少额外的耦合。工程师是否可以轻松地在该库或框架之外编写和测试代码，又或者新的库或框架是否需要配备额外的运行环境，而拖慢开发周期？

除了选择新的库和框架以外，审查代码需要考虑代码变更对未来变更的支持。如果系统某处突然需要一个额外的集成点，并且该集成点将会变化，那么需要涉及多少处更新呢？当然，开发人员必须留意过度设计，避免因为变更而永久地增加额外的复杂度或抽象。《重构》一书提供了相关建议。

> **事不过三，三则重构**
>
> 第一次做某件事时，你尽管去做。第二次做类似的事时，你会对重复犹豫，但无论如何你还是重复这件事。第三次再处理类似的事时，你应该将其重构。

交付新功能是驱动和奖励团队最常见的原因，但对于代码质量和演进性方面，只有在团队重视时才会予以考虑。负责演进式架构的架构师需要注意团队的行为，优先考虑那些有助于或支持演进能力的设计决策。

8.2.2 试验文化

成功的演进离不开试验，但有些公司因为忙于交付而无暇进行试验。成功的试验是经常进行一些小型活动来尝试新的想法（从技术和产品角度）并将成功的试验集成到现有系统中。

> 衡量成功的真正标准是在 24 小时内能完成的试验数量。
>
> ——Thomas Alva Edison

组织可以通过如下几种方式鼓励试验。

- 从外部吸收想法

  很多公司派员工参加展会，并鼓励他们寻求新的技术、工具和能更好地解决问题的方法。还有公司将外部建议或顾问作为新想法的来源。

- 鼓励明确的改进

  丰田公司因其持续改善（kaizen）文化而闻名。期望每个人能不断地寻求持续改善，特别是那些最了解问题并负责解决问题的人。

- 进行探针试验并稳定下来

  探针试验是极限编程实践，让团队构建一个临时方案以快速了解某个棘手的技术问题、探索某个不熟悉的领域，或者提升估算的信心。使用探针试验会牺牲软件质量来提升学习速度。没人会把探针试验得到的方案直接用于生产环境，因为它缺少必要的考量和时间来使其切实可行。它是为学习而生的，不是精心设计的方案。

- 创造创新时间

  谷歌因其“20% 的时间”闻名于世，其员工可以将其 20% 的工作时间用于任意项目。其他公司组织黑客马拉松并允许团队探索新产品或改进现有产品。Atlassian 定期召开 24 小时会议，并称其为 ShipIt。

- 采用基于集合的开发方式

  基于集合的开发专注于探索多种方法。乍一看，由于额外的工作，多个可选项很费功夫，但在探索多个选项的同时，团队最终能更好地理解问题，并通过工具和方法找到真正的约束。令该方法有效的关键是在短时间（几天）内构建多个原型，从而获取更具体的数据和体验。在综合考虑多个竞选方案后，往往才能得出更好的方案。

- 连接工程师和最终用户

  只有当团队清楚试验的影响，试验才会成功。在许多具有试验思维的企业里，团队和产品人员能直接看到决策对最终用户的影响，并被鼓励通过试验来探索这种影响。A/B 测试是企业应用这种试验思维的实践。企业的另一种实践是派团队和工程师观察用户是如何与软件交互来完成某项任务的。这种实践源于可用性社区的文章，让工程师了解最终用户的感受，以更好地理解用户需求，并通过新的想法来更好地满足他们。

8.3 首席财务官和预算

在演进式架构中，企业架构的一些传统功能必须反映不断变化的优先级，例如预算。过去，在软件开发领域，预测长期趋势的能力是预算的基础。然而，正如本书所建议的，动态平衡的基本特质使其无法被预测。

事实上，在架构量子和架构成本之间存在着有趣的联系。随着架构量子数量的增加，每个架构量子的成本降低，直到达到最佳点，如图 8-3 所示。

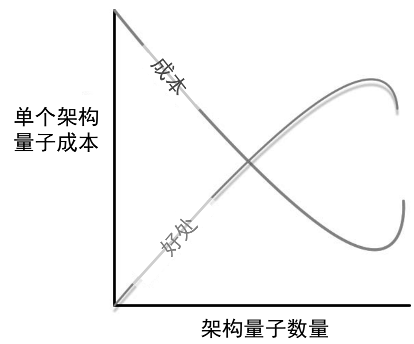


图 8-3：架构量子和成本之间的关系

在图 8-3 中，随着架构量子数量的增加，每个量子的成本由于多个因素减少了。首先，由于架构由较小的部分组成，问题需要分离得更加离散和明确。其次，物理量子数量的增加需要运维方面的自动化，因为当量子数量超过某个点后，人们将无法再手动处理这类事务。

然而，过小的架构量子有可能使绝对数量的成本高昂。例如，在微服务架构中，构建服务的粒度可以细到表单中的每个字段。在这样的粒度下，各个细小部分间的协调成本开始主导架构中的其他因素。因此，在图中的极端情况下，架构量子的绝对数量导致每个架构量子能获得的好处减少。

在演进式架构中，架构师追求合适的量子大小和对应成本之间的最佳点。每个公司情况各异。例如，市场迅猛发展，公司可能需要更快的变更速度，因此需要更小的架构量子。记住，新一代架构的出现速度与生产周期成正比，架构量子越小，生产周期越短。

有些公司可能发现构建架构量子更大的、基于服务的架构更实用（见第 4 章），因为它能更容易模拟常规变更。

由于我们面对着无视规划的体系，架构和成本的最优配置取决于很多因素。这也反映了我们所观察到的，架构师的角色外延了，架构选择比以往任何时候都更具影响力。

与其遵循老旧的企业架构“最佳实践”指南，现代架构师必须明白演进式架构的长处及其固有的不确定性。

8.4 构建企业适应度函数

在演进式架构中，企业架构师的工作主要围绕着架构指导和企业级适应度函数展开。微服务架构反映了这一模式转变。由于在运维上各个服务彼此分离，所以不再需要考虑资源共享。相反，架构师指导架构中有明确目标的耦合点（例如服务模板）和平台选择。企业架构师通常负责共享基础设施功能，以及为了企业内部的一致性，将平台选择限制在一定范围内。

> **案例分析：开源库的合法性**
>
> 有一次，PenultimateWidgets 的律师开始质疑公司使用开源库的合法性。他们仔细阅读了每个框架和库的使用许可，最终确定 PenultimateWidgets 没有使用任何可能引发问题的开源库。但有律师问到："我们如何知晓许可条例是否发生了变化呢？" PenultimateWidgets 没有这样的服务。
>
> 为了解决这一问题，法务团队在证实了当前开源库的合法性后，开发人员就在库中找到使用许可的内容并创建了一个**临时适应度函数**，用它来持续检查该文件内容的变化。因此，每次开源库的许可发生变化（无论什么原因），适应度函数便会触发并通知这一变化。当然，适应度函数不会智能到能够确定变更是否合适，仍然需要人工应对这些琐事，但架构师能构建适应度函数来引发定向关注，而不是自动化的解决方案。

演进式架构赋予企业架构师的另一个新职责是定义企业级适应度函数。企业架构师通常负责企业级非功能需求，例如伸缩性和安全性。很多组织缺乏自动评估能力来评估在单个项目和总体上这些架构特征的表现。一旦项目采用了适应度函数来保护架构的各个部分，企业架构师可以利用相同的机制验证这些企业级的特征保持不变。

如果每个项目都使用部署流水线来将适应度函数应用于其构建中，企业架构师也可以在其中插入一些自己的适应度函数。这使得每个项目可以持续校验横切关注点，例如伸缩性、安全性及其他企业级问题，尽早发现缺陷。正如微服务项目共享服务模板可以统一技术架构，企业架构师可以使用部署流水线来推动跨项目的一致性测试。

案例研究：PenultimateWidgets平台化

PenultimateWidgets 的业务发展得很好，他们决定将部分平台卖给其他小商品卖家。PenultimateWidgets 平台的吸引力部分来自于其经过验证的伸缩性、回弹性、性能等优点。然而，由于用户数量急剧增长，他们的架构师不希望平台刚开始出售就以失败结束。

为了维护平台的重要架构特征，PenultimateWidgets 的架构师随平台提供了部署流水线，以及围绕重要架构维度构建的适应度函数。平台用户必须维持已有的适应度函数才能合法地使用平台，以及在扩展平台时添加自己的适应度函数。

8.5 从何开始

很多处理现有架构（比如大泥团架构）的架构师努力找寻为架构增添演进能力的起点。虽然适当的耦合和应用模块化通常是你应该采取的第一步，但有时还有其他优先事项。例如，如果你的数据模式糟糕地耦合在一起，那么确定 DBA 如何实现模块化可能是第一步。以下是一些构建演进式架构实践的相关通用策略和理由。

8.5.1 容易实现的目标

如果组织需要早期成功来证明这种方法，架构师可以选择最简单的问题来凸显演进式架构方法。通常，这是系统中在很大程度上已经解耦的一部分，而且最好不在任何依赖的关键路径上。团队可以通过增强模块性和降低耦合来展示演进式架构的其他方面，如适应度函数和增量变更。构建更好的隔离可以使测试和适应度函数更具针对性。更好地隔离可部署单元使得构建部署流水线更容易，并为构建更强大的测试提供了平台。

在采取增量变更的环境中，衡量指标常附属于部署流水线。如果团队通过指标数据进行概念验证，开发人员应该在验证前后收集适当的指标数据。收集具体数据是开发人员审查其方法的最佳方式，记住**实证胜于雄辩**。

这种“最简单者优先”的方法将风险降到了最低，但可能牺牲价值，除非团队有幸找到**既容易解决**、**价值也高**的问题。对于那些持怀疑态度并想试水演进式架构的公司来说，这是很好的策略。

8.5.2 最高价值优先

除了“最简单者优先”外，另一种方法“最高价值优先”找到系统中最关键的部分，围绕它构建演进行为。公司可能出于以下几个原因采取该方法。第一，如果架构师确信要实现演进式架构，那么选择价值最高的部分就表明了决心。第二，对于那些仍在评估想法的公司，他们的架构师可能对这些技术在体系中的适用性感兴趣。因此优先选择价值最高的部分，便能明确演进式架构的长远价值。第三，如果架构师怀疑这些方法的适用性，那么用系统中最有价值的部分来审查这些概念，能够为是否继续提供了可行的数据。

8.5.3 测试

很多公司苦于系统缺乏测试。如果开发人员发现他们的代码库缺乏测试，在向演进式架构做出更具大胆的行动前，他们会添加一些关键测试。

通常，管理层不赞成开发人员进行只为代码库添加测试的项目。他们对这类活动持怀疑态度，特别是新功能无法如期实现时。于是，架构师应该将模块化增强和高级功能测试结合起来。用单元测试封装功能来为测试驱动开发（TDD）等工程实践提供更好的基础。但更

新代码库需要时间，因而在重建代码之前，开发人员应该对一些行为添加粗粒度的功能测试，以验证系统的总体行为不会因为重建而改变。

对演进式架构的增量变更而言，测试是关键的组件，并且适应度函数也在积极地利用测试。因此，至少在某种程度上，测试使这些技术成为可能，而且实现演进式架构的难易程度和测试的综合性密切相关。

8.5.4 基础设施

对一些公司而言，构建新能力需要时间，而运维团队常受困于缺乏创新。对那些基础设施功能失调的公司来说，构建演进式架构之前可能需要先解决这些问题。基础设施问题通常有很多形式。例如，有些公司将所有的运维工作外包给别的公司，因此无法控制其体系的关键部分。当需要承担跨公司协调的额外开销时，DevOps 的难度呈级数式上升。

另一个常见的基础设施功能失调是开发和运维之间无法穿透的防火墙，因为开发人员根本不了解代码最终将如何运行。这种结构在部门间充斥着权力博弈的公司里很常见，因为每个团队都各行其是。

最后，在某些组织中，架构师和开发人员都忽视好的实践，而不断引入大量技术债，这些技术债体现在基础设施中。一些公司甚至连运行环境和运行主体都不清楚，也不了解架构和基础设施之间交互的基本知识。

基础设施能影响架构

Neal 曾为某个服务托管公司提供咨询工作。该公司管理大量服务器（当时约有 2500 台），运维团队被划分成了三个小团队：一个团队安装硬件，另一个安装操作系统，第三个安装应用程序。无须多言，当开发人员需要某个资源时，他们向运维的黑洞中提交一个工单，接着会生成更多工单并来回折腾几周才准备好资源。不巧的是，CIO 在一年前离开了公司，之后 CFO 接手了该部门。当然，CFO 主要关注成本节约，而不是将他视为仅仅是开销的那部分工作现代化。

在调查运维的薄弱环节时，一位开发人员指出，每个服务器只能容纳大约 5 位用户，鉴于系统的简单程度，这实在是令人咋舌。开发人员羞怯地解释道，HTTP 会话状态的滥用到了前所未有的程度，最终使其成为了一个巨大的内存数据库。因此，每个服务器只能为几个用户提供服务。运维团队无法创造一个像样的类生产环境用于调试，而且出于权责原因，他们也绝对不允许开发人员在生产环境中进行调试（或进行大量监控）。因为无法和实际运行的应用版本交互，开发人员无法化解他们逐渐造成的混乱局面。

经过粗略的计算，我们确定了公司运行的服务器数量可以减少一个数量级，到 250 台。但公司却一直忙于购买新的服务器、安装操作系统等。当然，更为讽刺的是，他们节省成本的措施事实上花费了公司一大笔钱。

> 最终，被围攻的开发人员组建了自己的 DevOps 团队，并开始自主管理服务器，完全绕过了传统的运营团队。两个团队间的斗争正在酝酿，但是短期内，开发人员开始重建他们的应用。

最终，这个建议和咨询师的烦人但精确的答案在某种程度上不谋而合。只有架构师、开发人员、DBA、测试人员、安全顾问和其他贡献者才能最终确定最佳的演进式架构路线图。

8.5.5 PenultimateWidgets的企业架构师

PenultimateWidgets 正在考虑改造旧平台的主要部分，企业架构师们用电子表格列出了新平台需要具备的所有特征：安全性、性能指标、伸缩性、可部署性等一系列特征。每个类别包含 5 到 20 个单元格，每个单元格都定义了一些特定的标准。例如，正常运行时间指标中有一条要求每个服务的可用性必须达到“5 个 9”（99.999%）。他们总共定义了 62 项。

但是他们意识到该方法的一些问题。首先，是否要验证每一项呢？他们可以制定一些原则，但是谁来持续验证这些原则呢？人工验证所有这些标准，即便是偶尔为之，也会是相当大的挑战。

其次，对系统的每个部分强加严格的可用性准则合理吗？使管理员的管理界面具备“5 个 9”的可用性真的至关重要吗？制定全面的原则往往会导致严重的过度设计。

为了解决这些问题，企业架构师们将这些标准定义为适应度函数，并为每个项目创建初始的部署流水线模板。在部署流水线上，架构师设计让适应度函数自动检查那些关键特性，例如安全性，让各个团队添加其服务特定适应度函数，比如可用性。

8.6 演进式架构的未来

演进式架构的未来会是什么？随着团队越来越熟悉这些想法和实践，他们会将其纳入日常工作中，根据这些想法构建新能力，例如数据驱动开发。

构建更复杂的适应度函数需要做很多工作，但是随着一些组织解决问题并开源他们的方案，事情正在变得容易。在早期的敏捷开发中，人们遗憾于一些问题难以自动化，但是无畏的开发人员披荆斩棘，如今整个数据中心都实现了自动化。例如，Netflix 在概念化和构建工具方面所做出的巨大创新、作为整体持续式适应度函数的 Simian Army（虽然还未称其为适应度函数）。

这里有几个有前景的领域。

8.6.1 基于AI的适应度函数

对普通项目来说，大型开源人工智能框架将逐渐变得可用。随着开发人员学着运用这些工具来支持软件开发，我们预想通过基于 AI 的适应度函数来查找反常的系统行为。信用卡企业已经开始了探索，例如标记出在世界不同地点几乎同时发生的交易。架构师可以通过构建调查工具来查找架构中的异常行为。

8.6.2 生成式测试

在很多函数式编程社区中，**生成测试**是广受欢迎的常见实践。传统单元测试包含对每个测试用例结果正确与否的判断。然而，通过生成测试，开发人员运行大量测试并抓取结果，然后对结果进行统计分析来查找反常的行为。例如最常用的边界值检查，传统单元测试检查已知的数字临界点（负数、不断增加的数字等），但无法覆盖意外的少数情况。生成测试检查每一个可能的数值并报告失败的少数情况。

8.7 为什么（不）呢

架构中没有灵丹妙药。如果无法从演进性中获益，我们不建议在项目中为其付出额外的成本和精力。

8.7.1 公司为何决定构建演进式架构

很多公司发现过去几年间变化正加速，正如前面《福布斯》所提到的，每个公司都必须具备软件开发和交付能力。

1. 可预测性与可演进性

很多公司看重对资源和其他战略问题的长期规划，显然，公司看重**可预测性**。然而，由于软件开发领域的动态平衡，事物变幻莫测。企业架构师可能还计划，但计划可能随时失效。

就算身处传统的、成熟行业的公司也不应忽视系统无法演进的风险。出租车行业是存在了数个世纪的国际化体系，当它受到共享汽车行业冲击时，它认识到形势已变并对变化的环境做出了反应。该现象正如《创新者的窘境》所预测的那样，成熟市场中的公司，失败的可能性更大，因为很多灵活的初创公司能更好地应对环境变化。

构建可演进的架构会耗费额外的时间和精力，但好处是公司可以应对市场的重大变化，而不需要大量返工。可预见的是，过去大型机和专用运维中心的日子一去不复返了。软件开发领域易变的本质将所有组织推向增量变更。

2. 规模

曾几何时，架构的最佳实践是构建基于关系型数据库的事务系统，利用大量数据库功能进行协调。如何伸展规模是该方式所面临的问题，后端数据库难以伸展。为了缓解该问题的影响，很多复杂的技术应运而生，但它们无法解决问题的根源——耦合。架构中任何耦合点最终将阻碍架构伸展，依赖数据库进行协调终将受到这一限制。

Amazon 曾面临相同的问题。最初站点设计为单体前端与围绕数据库设计的单体后端连接。当流量增大时，他们不得不伸展数据库。当数据库的规模达到上限，其站点的性能随之下降，每个页面的加载速度都变慢了。

Amazon 意识到，将所有事物耦合于某个组件（无论是关系型数据库、企业服务总线，或是其他）最终将限制系统的规模。通过将架构重新设计为微服务式来消除不当的耦合，整个系统得以伸展。

这种程度的解耦带来的额外好处是强化了可演进性。正如本书所介绍的，不当的耦合是演进的最大挑战。因此，构建可伸缩的系统相当于构建可演进的系统。

3. 高级业务能力

很多公司眼红 Facebook、Netflix 和其他顶尖的技术公司，因为它们拥有先进的功能。增量变更支持一些熟知的实践，例如研究假设和数据驱动开发。很多公司渴望通过多元测试将用户纳入其反馈环中。可演进的架构是许多高级 DevOps 实践的关键基础。例如，一旦组件高度耦合，使问题难以分离，开发人员将难以执行 A/B 测试。通常，在面对不可避免又无法预测的变化时，演进性架构能赋予公司更好的技术响应能力。

4. 以生产周期为业务指标

3.1.2 节区分了持续交付和持续部署。前者在部署流水线中至少有一个阶段进行了人工拉取[6]。而后者每个阶段都在验证成功后自动升级到下个阶段。构建持续部署需要一定的工程技巧，但是为什么企业需要达到这个程度呢？

因为在某些市场中，生产周期已经成为了一种业务差异。一些大型传统组织将软件视为开销，并缩减其支出。创新型公司则将软件视为竞争优势。例如，如果 AcmeWidgets 的架构的生产周期为 3 小时，而 PenultimateWidgets 的生产周期依旧是 6 周，那么 AcmeWidgets 就具备了可利用的优势。

很多公司已将生产周期作为头等业务指标，主要是因为他们所处的市场竞争激烈。所有市场最终将变得竞争激烈。例如，20 世纪 90 年代早期，一些大型公司倾向于通过软件将手动工作流自动化，且获得了巨大优势，最终所有公司都意识到了这样做的必要性。

注 6：通常是在做出发布决策时。——译者注

5. 在量子级别隔离架构特征

将传统非功能性需求视为适应度函数，并构建高内聚的架构量子让架构师可以支持每个架构量子的不同特征，这也是微服务架构的优点之一。因为各个架构量子的技术架构彼此解耦，架构师可以根据不同情况选择不同架构。例如，在小型设备上工作的开发人员可能选择微内核架构，因为他们想支持允许增量添加的小核心。为了伸缩性，另一个开发团队可能选择事件驱动架构来构建服务。如果它们都是某个单体应用的一部分，架构师需求权衡这些需求。通过在架构量子级别隔离技术架构，架构师可以专注于单个量子的主要特征，而无须权衡竞争优先级。

8.7.2 案例分析：PenultimateWidgets选择性伸展

PenultimateWidgets 的某些服务不太需要伸缩，并因此使用简单的技术栈编写。然而，有几个服务凸显出来。导入服务需要每晚导入实体店的库存数据到会计系统。因此，开发人员为导入服务构建的架构特征及适应度函数中包括了**伸缩性**和**回弹性**，使得该服务的技术架构变得极其复杂。另一个服务叫作市场提要（`MarketingFeed`）服务，每个店铺在每天营业前都会调用它来获取销售和市场更新。操作上，市场提要服务需要**弹性**才能在不同时区的店铺开始营业时处理爆发的请求。

无意的过度设计是高度耦合的架构中的一个普遍问题。在更耦合的架构中，开发人员不得不将伸缩性、回弹性和弹性构建到每个服务中，使得不需要这些能力的服务复杂化。架构师习惯于在权衡后选择架构。构建具有清晰量子边界的架构使架构师可以精确指定所需的架构特征。

适应还是演进

很多组织陷入技术债不断增加且不愿意进行必要重建的困境，导致系统和集成点变得日益脆弱。有的公司尝试将服务总线等连接工具作为弥补措施，虽然减轻了一部分技术问题，但并没有解决业务流程更深层次的逻辑内聚。使用服务总线是让现有系统**适应**另一种设置的例子。但前面强调过，适应的副作用是技术债增加。当开发人员适应某些东西，他们会保留原始行为并引入新的行为与之并行。组件经历的适应次数越多，并行的行为就越多，也就越复杂，最好是具有战略意义的。

功能开关的使用很好地展示了适应的好处。通常，在通过假设驱动开发尝试不同的交互方案来测试用户时，开发人员会使用功能开关来寻求最佳方案。在这种情况下，功能开关引入的技术债是有目的且可取的。当然，这类开关的最佳工程实践是在做出决策后尽快将其删除。

另外，**演进**意味着根本性的变化。构建可演进的架构需要就地改变架构，设置适应度函数以防止架构被破坏。最终的结果是系统持续以有效的方式演进，并且过时的方案不会遗留其中。

8.7.3 企业为何选择不构建演进式架构

我们认为演进式架构不能解决所有问题。企业有几种合理的理由这样的想法。

1. 大泥团无法演进

架构师可能忽略的一个关键特征是**可行性**——团队是否应该承担该项目？如果架构是糟糕地耦合在一起的大泥团，显然，赋予其演进能力将耗费大量的工作，可能比从头重写的工作量还要大。公司不愿抛弃任何具有感知价值的东西，但通常，返工比重写的成本更高。

如何判断公司是否处于这种状况？将现有架构转变为可演进的架构的第一步是**模块化**。因此，开发人员的首要任务是研究当前项目是否具有模块性，并围绕这些发现重建架构。一旦架构变得不那么杂乱，架构师便能更轻松地了解底层结构并为重建做出合理的决策。

2. 其他架构特征占主导地位

在选择特殊的架构样式时，**演进性**只是架构师需要权衡的众多特征之一。没有架构可以完全支持冲突的核心目标。例如，同一个架构中高性能和高伸缩性很难并存。在某些情况下，有的因素可能比演进式变更更重要。

大多数时候，架构师为了各种需求选择架构。例如，某个架构需要支持高可用性、安全性和伸缩性，那么一般的架构模式是适用的，例如单体架构、微服务架构或 EDA。然而，有一些领域专用架构试图将单个特征最大化。

LMAX（一种自定义交易解决方案）是领域专用架构的绝佳例子。其主要目标是实现快速的交易吞吐量，他们尝试了多种技术都但没有成功。最终，通过分析架构最底层，他们发现伸缩性的关键是让其逻辑小到适应 CPU 缓存，并预先分配内存来防止垃圾回收。他们的架构在单个 Java 线程上实现了惊人的每秒 600 万次交易！

为了特定目的构建的架构，如果为了适应其他问题而进行演进会很困难（除非开发人员特别幸运地发现架构问题是重合的）。因此，大多数领域专用架构不关注演进性，因为其特定目的优先于其他问题。

3. 牺牲架构

Martin Fowler 将牺牲架构定义为“可抛弃的架构”。很多公司需要在开始时构建简单版本来进行市场调查或验证可行性。一旦验证成功，他们可以构建**真正**的架构来支持已经显现的架构特征。

很多公司战略性地采取该方法。通常，公司在创建最小可行性产品来测试市场时会采用此类架构，并打算在获得市场认可后再构建更强大的架构。构建牺牲架构意味着架构师不会尝试演进它，而会在适当的时候用更长期的方案替代它。对于想探索新市场或产品可行性的公司而言，云上服务使牺牲架构成为了具有吸引力的选择。

4. 计划即将停止业务

演进式架构有助于企业应对环境变化。如果公司不准备在一年内开展业务，就没有理由构建可演进的架构。

有些公司正处于这个情况，只是他们还没意识到。

8.7.4 说服他人

架构师和开发人员希望非技术人员和管理层理解演进式架构的好处。当组织的某些部分被必要的变更扰乱时尤为如此。例如，当开发人员指出运维部门工作不当时，通常会遇到阻力。

第 6 章介绍过解决这类问题的最佳方案。与其尝试说服组织中的保守人群，不如展示这些想法对其实践的改进。

8.7.5 案例分析："咨询柔道"

有位同事曾和某大型零售商共事，他尝试说服企业架构师和运维团队采用更现代的 DevOps 实践，例如自动配置服务器和更完善的监控等。然而对方对他的请求置若罔闻，常有这两种托词："我们没有时间""我们的配置太过复杂，这种东西是不会奏效的"。

他采用了一种被称为**咨询柔道**的高明技巧。柔道作为一种武术，有很多技巧是利用对手的重量进行对抗。咨询柔道则需要找到一个特定的痛点，并将解决该痛点作为案例。该零售商的痛点在于缺乏足够的质量保障环境。因此，团队尝试共享环境，但这带来了极大的困扰。在给出案例分析之后，他获准使用现代 DevOps 工具和技术来构建质量保障环境。

当完成后，他推翻了之前的两个错误假设。现在，任何团队都能轻松配置所需的质量保障环境。他所做出的工作由于价值显著，说服了运维团队投入更多资源到现代技术中。实证胜于雄辩。

8.8 商业案例

业务人员通常对雄心勃勃的 IT 项目格外谨慎，因为它们听起来像是花大价钱更换水管。然而，很多企业发现更具演进性的架构能为他们所渴望的能力提供基础。

8.8.1 未来已来……

> 未来已来，只是尚未流行。
>
> ——William Gibson

很多公司将软件视为开销，就像供热系统。当软件架构师与公司高管谈论软件创新时，公司高管会觉得像是管道工在以高昂的价格推销他们的工作。然而，对软件的战略重要性持怀疑态度有损形象。因此，软件采购的决策者往往在制度上变得保守，更看重节约成本而不是创新。企业架构师犯这种错误是能够理解的，因为他们会参考同类公司，看他们是如何做出这些决定的。但是这种做法非常危险，因为拥有现代软件架构的公司可能颠覆性地进入现有的业务领域，并快速占领市场，因为他们的信息技术更先进。

8.8.2 没有后顾之忧地快速前行

很多大型企业都在抱怨组织内的变化步伐缓慢。构建演进式架构的优势在于更高的工程效率。所有被称为**增量变更的**实践都会提高自动化和效率。将最顶层的企业架构问题定义为适应度函数，不仅将一组完全不同的问题统一到一处，还迫使开发人员根据客观结果考虑架构。

构建演进式架构意味着团队有信心在架构层开展增量变更。第 2 章介绍了 GitHub 的案例，其中对架构基础组件的变更没有引起任何回归错误（并且发现了潜藏的错误）。业务人员害怕破坏性的变更。如果开发人员构建允许增量变更、比旧架构更强大的架构，将会带来业务和工程上的双赢。

8.8.3 风险更低

随着工程实践的提高，风险会降低。为了实现架构演进和增量变更，团队不得不采用现代实践，这是其优点之一。一旦开发人员有信心在不破坏任何事物的情况下改变架构，那么公司便能构建更多发布候选版本。

8.8.4 新能力

向业务人员推荐演进式架构的最佳方式是介绍它的新业务能力，例如假设驱动开发。当架构师用专业术语描述技术改进时，业务人员会一脸茫然，因此最好用业务语言来说服他们。

8.9 构建演进式架构

我们关于构建演进式架构的想法基于并依赖于很多已有的实践建立，例如测试、指标、部署流水线和其他基础设施及创新。我们通过适应度函数这一新的视角统一了先前的各种概念。

对我们而言，任何校验架构的东西都是适应度函数，同等对待这些验证机制使得自动化和验证更加容易。 我们希望架构师开始将架构特征视为可评估的事物，而非突发奇想，这样他们能构建出更具回弹性的架构。

使系统更具演进性并不简单，但我们没有其他选择。软件开发环境将不断地产生意想不到的新想法。那些能够在这种环境中做出反应并成长的组织将获得极大的优势。

关于作者

Neal Ford（尼尔·福特）是 ThoughtWorks 的技术总监、软件架构师及意见领袖。ThoughtWork 是一家软件公司，聚集了一群充满激情、以目标为导向的人，他们以极具创新的思考通过技术应对最困难的挑战，同时追求 IT 行业革新并倡导积极的社会变革。在加入 ThoughtWorks 之前，Neal 是 The DSW Group（美国一家知名的软件人才培养与软件开发公司）的技术总监。

Neal 毕业于佐治亚州立大学的计算机科学专业，他专攻语言与编译器，同时还辅修了统计分析相关数学。Neal 是国际公认的软件开发和交付专家，特别是在敏捷工程技术及软件架构领域。Neal 为杂志撰写过文章、出版了 7 本书、发表了许多视频演讲，还在全球 100 多场开发者大会上演讲过。其作品主题涵盖了软件架构、持续交付、函数式编程、前沿软件创新，以及以业务为重点来改进技术展现的图书和视频。他的咨询工作主要针对大型企业应用的设计和构建。

Rebecca Parsons（丽贝卡·帕森斯）博士是 ThoughtWorks 的首席技术官，拥有数十年的软件应用开发经验，涉及多个行业和系统。她的技术经验包括领导创建大规模分布式应用、不同系统的集成以及和架构师团队共事。除了对深层次技术充满热情外，她还是技术行业性别多元化的坚定拥护者。

在加入 ThoughtWorks 之前，Parsons 博士在中佛罗里达大学担任计算机科学专业的助理教授，教授编译器、程序优化、分布式计算、编程语言、计算理论、机器学习和计算生物学等课程。她还在洛斯阿拉莫斯国家实验室担任过主任博士后研究员，研究并行和分布式计算、遗传算法、计算生物学和非线性动力系统等问题。

Parsons 博士在布拉德利大学获得了计算机科学与经济学学士学位，在莱斯大学获得了计算机科学硕士学位和博士学位。她还是 *Domain-Specific Languages*、《ThoughtWorks 文集》和《演进式架构》作者之一。

Patrick Kua（帕特里克·柯）是 ThoughtWorks 前首席技术顾问，在技术行业工作超过 15 年。他在通过寻求技术、人员和流程之间的平衡来提高软件交付效率方面享有盛名。他还在许多会议上就技术领导力、架构和建立强大的工程文化等话题发表过演讲。

Patrick 著有 *The Retrospective Handbook: A Guide for Agile Teams* 和 *Talking with Tech Leads: From Novices to Practitioners*，他还建立了定期培训项目来支持开发人员向技术主管或架构师转变。

封面介绍

本书封面上的动物是八字脑珊瑚，也称“褶皱”珊瑚或“火山口”珊瑚，原生于印度洋。这种珊瑚以其独特的褶皱、鲜艳的体色和耐寒性而闻名。白天它们依靠虫黄藻光合作用的产物生存，夜里它们伸出触手将猎物引诱到口中（有些八字脑珊瑚有两到三张口），其中包括浮游生物和小鱼。

由于外表引人注目、适应性强，八字脑珊瑚是水族馆的热门选择，它们能在水族馆水底的泥沙中生活，这类似于其原生栖息地的浅海底。它们喜欢水流适中且动植物丰富的环境。

IUCN（世界自然保护联盟）红色名录将八字脑珊瑚列为近危物种。O’Reilly 图书封面上的许多动物都濒临灭绝，所有这些动物对世界都很重要。要了解有关如何提供帮助的更多信息，请访问 animals.oreilly.com。

封面图片来自 Jean Vincent Félix Lamouroux 的 *Exposition Methodique des genres de L’Ordre des Polypiers*。

O'REILLY®

演进式架构

企业架构师不能再依赖静态计划了。软件开发体系在持续变化，新的工具、框架、技术和范式不断涌现。这给脆弱的系统带来了挑战，但也提供了更好的解决方案。近年来，核心软件工程实践中的快速变化让我们重新思考如何更改架构，使其与时俱进。本书结合相关实践，给出了让架构适应变化的新思路。

构建演进式架构主要涉及3个方面：适用度函数、增量变更和适当的耦合。ThoughtWorks的3位专家各讲一个方面，然后综述如何构建支持持续变更的架构。

- **适应度函数：架构呈现或前进的目标**
- **增量变更：在开发和运维中实现渐进改变**
- **架构耦合：确定适当的架构耦合以支持无瑕变更**
- **演进式数据：随时间推移按要求和架构转变演进数据库**
- **构建可演进的架构：结合以上各方面构建演进式架构**
- **实践演进式架构：助你起步的实践指南**

尼尔·福特（Neal Ford）是ThoughtWorks软件架构师、Meme Wrangler，持续集成与交付领域专家。

丽贝卡·帕森斯（Rebecca Parsons）是ThoughtWorks CTO，在大规模分布式对象应用开发和集成方面拥有丰富经验。

帕特里克·柯（Patrick Kua）是ThoughtWorks前首席科学家，在敏捷和精益开发方面拥有丰富经验。

“这本书着眼于理论、立足于实践，架构师们定会受益匪浅。几十年前我若能读到它该多好，当然现在也不晚。”

——Venkat Subramaniam博士

Agile Developer公司创始人

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn

热线：(010)51095183转600

分类建议　计算机/软件开发/微服务

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-51617-6

ISBN 978-7-115-51617-6

定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks